Hochschule
Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Asymmetric Low-power FHSS Algorithm

Master Thesis

Study program:	N2612 – Electrical Engineering and Informatics
Study course:	3906T001 – Mechatronics
Author:	**Tomáš Jakubík**
Assessor:	Prof. Dr.-Ing. Dietmar Scharf (Hochschule Zittau/Görlitz)
Supervisor:	Ing. Jaromír Šubčík (Jablotron Alarms a.s.)

Zittau/Liberec 2016

Faculty of Electrical Engineering and Computer Science

Hochschule Zittau/Görlitz
UNIVERSITY OF APPLIED SCIENCES

Assignment for the master thesis

Course of studies: Mechatronics

Student's name: Tomas Jakubik

Subject:

Asymmetric low-power FHSS algorithm

Assignment:

- Explore common FHSS technologies
- Design an FHSS algorithm satisfying following conditions
  - There are grid powered HUBs and battery powered peripherals
  - HUBs do not communicate with each other over the FHSS network
  - Response time from peripheral to the HUB must be less than 400 ms
  - Peripherals are monitored within 20 minutes
  - Basic modulation is GFSK on CC1200 transceiver at an SRD frequency band h1.1
- Compare achieved parameters with a non-hopping solution

Assessor: Prof. Dr.-Ing. Dietmar Scharf (Hochschule Zittau/Görlitz)

Supervisor: Ing. Jaromir Šubčik (Jablotron Alarms a.s.)

Date of issue: 12.04.2016

Date of submission: 12.08.2016

Registry-Number.: MA/EMIm14 – 07/ 16

Assessor

Dean

Declaration

I hereby certify that I have been informed that Act 121/2000, the Copyright Act of the Czech Republic, namely §60, Schoolwork, applies to my master thesis in full scope. I acknowledge that the Technical University of Liberec (TUL) does not infringe my copyrights by using my master thesis for TUL's internal purposes.

I am aware of my obligation to inform TUL on having used or licensed to use my master thesis in which event TUL may require compensation of costs incurred in creating the work at up to their actual amount.

I have written my master thesis myself using literature listed therein and consulting it with my supervisor and my tutor.

I hereby also declare that the hard copy of my master thesis is identical with its electronic form as saved at the IS STAG portal.

Date:

Signature:

Acknowledgements

I would like to thank the company
Jablotron Alarms a.s.
for giving me material and intellectual support throughout
this thesis.

Special thanks come to both universities
Technical University of Liberec
and
Hochschule Zittau/Görlitz
for the opportunity to study such an interesting course.

Abstract

English

This thesis documents development and basic testing of an asymmetric FHSS network. Asymmetric in a way that the network HUB or master consumes a lot of power, while the peripherals or slaves consume as little power as possible. The basis of the network are simple GFSK transceivers CC1200 produced by TI. The intended bandwidth is an SRD band h1.1 commonly known as 868 MHz band. The controller MCUs used in this thesis are STM32F0 and STM32L0. Both are Cortex-M ARM processors produced by ST.

The created network enables the connection of input only peripherals with input response time of 400 ms, while keeping the cumulative current consumption of 4.27 µA. Peripherals receiving in periodic intervals were also discussed. The principles explored in this thesis can be used with different transceivers, different modulations or different MCUs. Some of the measurements are specific for the used hardware, but the results should be easy to extrapolate to any platform of interest.

The thesis doesn't discuss PCBs development or development of any hardware at all.

Czech

Tato práce dokumentuje vývoj a základní testování asymetrické FHSS sítě. Asymetrické takovým způsobem, že HUB, neboli master, sítě spotřebovává mnoho elektrické energie, zatímco periferie, neboli zařízení slave, spotřebovávají energie co nejméně. Základem sítě jsou jednoduché GFSK přijímače CC1200 vyráběné společností TI. Uvažované frekvenční pásmo je SRD pásmo h1.1 známé také jako pásmo 868 MHz. Použité MCU jsou STM32F0 a STM32L0. Oba jsou Cortex-M ARM procesory vyráběné společností ST.

Vytvořená síť umožňuje připojení vstupních periferií s časem odezvy vstupu 400 ms přičemž kumulativní spotřeba proudu je jen 4.27 µA. Periferie přijímající v pravidelných intervalech byly také uvažovány. Principy použité v této práci mohou být použity s jinými přijímači, jinými modulacemi či jinými procesory. Některá z měření jsou specifická pro daný hardware, ale výsledky by měly být lehce extrapolovatelné na jakoukoli jinou platformu.

Tato práce se nezabývá návrhem PCB ani návrhem jakéhokoli hardware.

Keywords

Wireless, Spread Spectrum, FHSS, Low Power, STM32L0

Table of Contents

Illustrations, Tables and Equations

Symbols and Abbreviations

Table 1: Symbols and Abbreviations

Abbreviation	Meaning
π/4-DQPSK	π/4 rotated differential encoded quaternary phase shift keying
8DPSK	Differential encoded 8-ary phase shift keying
AFA	Adaptive frequency agility
b	Bit (unit of data length)
B	Byte (unit of data length)
BCD	Binary coded digital
BR	Basic rate (Bluetooth)
BT	Bandwidth bit product (GFSK filter)
CA	Collision avoidance
CCA	Clear channel assessment
CEPT	European Conference of Postal and Telecommunications Administrations
CRC	Cyclic redundancy check
CS	Carrier sense
CSMA	Carrier sense multiple access
DMA	Direct memory access
DSSS	Direct sequence spread spectrum
ECC	Electronic Communications Committee
ECO	European Communications Office
EDR	Enhanced data rate (Bluetooth)
EFIS	European Frequency Information System
ERP	Effective radiated power
FHSS	Frequency hopping spread spectrum
FIFO	First in first out
FSK	Frequency shift keying
GFSK	Gaussian frequency shift keying
GMSK	Gaussian minimum shift keying
GPIO	General purpose input output
HW	Hardware
IDE	Integrated development environment
ISM	Industrial, scientific, medical
LBT	Listen before talk
LCG	Linear congruential generator
LE	Low energy (Bluetooth)
LED	Light-emitting diode
LM	Link management
LQI	Link quality indicator (CC1200 variable)
MCU	Microcontroller unit
NVIC	Nested vectored interrupt controller
OOB	Out of band (emissions)
PAL	Protocol adaptation layer (Bluetooth)
PAN	Personal area network

Table 1: Symbols and Abbreviations (continued)

Abbreviation	Meaning
PCB	Printed circuit board
PLL	Phase locked loop
PNx	Pseudorandom (sequence)
PQT	Preamble quality threshold
RAM	Random access memory
RBW	Resolution bandwidth (spectrum analyser parameter)
RF	Radio frequency
RMS	Root-mean-square (mathematical analysis)
RTC	Real time clock
RX	Receiving, Receiver
SPI	Serial peripheral interface
SRD	Short range devices
SWD	Serial wire debug
TX	Transmitting, Transmitter
UART	Universal asynchronous receiver/transmitter
VTOR	Vector table offset register
WFI	Wait for interrupt (processor instruction)
XOR	Exclusive OR

1 Introduction

In the modern electronic development, there is a strong push towards spread spectrum radio communication. Spread spectrum technologies are better in resisting interference caused by nature, environment geometry, other radio equipment or deliberate jamming. The downside of spread spectrum technologies is the price or the firmware complexity.

One of the methods to spread the spectrum is a Frequency Hopping Spread Spectrum (FHSS) which uses sequences of pseudorandom hops. A part of the information is transmitted on one channel. Both transmitter and receiver hop to a different frequency, and a different part of the information is transmitted. Frequency of the hops varies from bits to whole packets.

In the case where the hops are made in-between packets, the hardware has enough time to settle on a different frequency. Every packet has a preamble helpful to synchronize the receiver on a given frequency. This means that an ordinary transceiver hardware used for a single frequency system can be used as well, and the additional price for the spread spectrum is zero.

1.1 The goal

The goal of this thesis is to create a demonstrative FHSS network for battery operated input sensors. Connection to or any direct interoperability with the Internet is not planned. Although this FHSS network may never be used ‘as is’, it will be an irreplaceable knowledge-base for anyone thinking about a simple FHSS solution on existing hardware or anyone selecting one of spread spectrum solutions.

1.2 Network concept

This is a star shaped wireless network with the HUB as a centre-point and multiple peripherals connected to it. The application layer of the HUB or its connection to superior system is not a topic of this thesis. In the following text I assume that it knows what to do and concentrate on the single cell wireless network. I noted the possibility that more than one network will exist at the same time and place and that they may or may not know of each other’s existence.

The HUB manages all the communication in the network, so in range of this thesis I called it the **master**. The opposite to master are peripherals which are called **slaves**.

The key parameter of the network is the response time from the slave to master, less than 400 ms, while keeping the battery consumption on a bearable level. To solve this problem, I let the input only slaves sleep for a maximal possible time. The loss of a slave should be noticed under 20 minutes. The slave can fully sleep for approximately 19 minutes and notify master about its presence in the remaining minute. After this time, the clock in master and slave cannot be synchronized.

The synchronization of a recently sleeping slave must be done sooner than 400 ms after an event. Slave cannot just start transmitting on an arbitrary channel because master would have to listen for that particular channel all the time. This is not a possibility on condition that master can receive only one channel at a time. My solution is that master sends beacon signals or **beacon packets** at short intervals. These beacon signals carry information needed to synchronize into the frequency hopping communication.

The beacon packets are sent on predetermined **beacon channels**. Slave remembers four different beacon channels before it goes to sleep. After the slave wakes up, it scans these four channels and receives the synchronization information on one of them. Master sends the beacon packet every 100 ms and each time on a different one of four beacon channels.

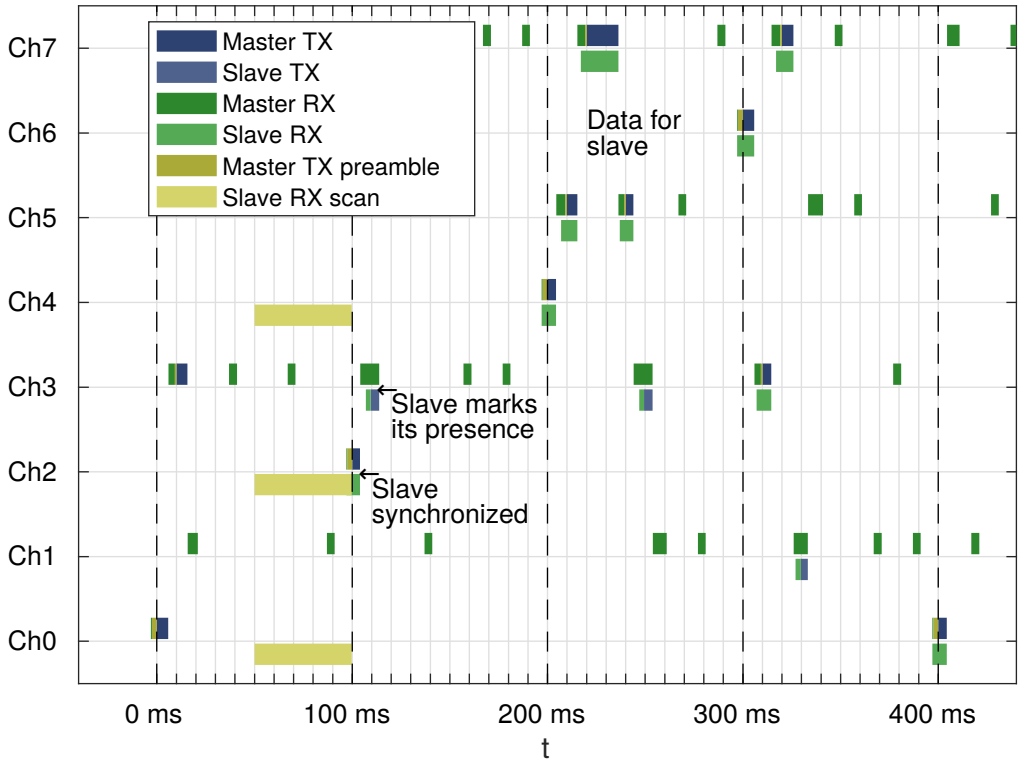

Illustration 1: Concept of the communication

There is a global view of the slave waking up from sleep on the illustration 1. Channels Ch0, Ch2, Ch4 and Ch6 are beacon channels. The remaining channels are used for the normal communication with synchronized frequency hopping. At a time of 50 ms, the slave wakes up. It scans the four beacon channels, and it receives the beacon packet at 100 ms. Slave learns the information about the next channel in the hopping sequence from the beacon packet, and transmits a simple packet marking its presence in the network. Master waits until time is 200 ms when it sends another beacon packet. Reason for this is described further in the thesis. Immediately after that, the master can transmit response or an arbitrary data for the slave on different frequencies in different packets.

2 The Hardware

Several Printed Circuit Boards (PCBs) were used for the initial development. The PCBs are fitted with the CC1200 [1], [2] RF transceiver and STM32F072 [3], [4] as a controller. The PCBs were designed at Jablotron Alarms for a different purpose. Apart from MicroContoller Unit (MCU) and RF chip, they are full of other components unused in this thesis.

STM32F072 is an entry-level MCU meaning that it is cheap and universal chip, but it is not particularly good in any of its parameters. Naming Cortex-M0, 32 bit, 48 MHz, processor as a 'basic' or 'entry-level' might be too much, but nowadays anything less than an ARM, apart from very specific situations, is outdated and too weak. The '2' at the end of the chip name means that it is not the simplest chip, but has more peripherals one of which is the USB. It was an important part in the original purpose of the board. The architecture Cortex-M0 [5] is perhaps the simplest ARM there is. It was developed from a Cortex-M3 by reducing the gate count and simplifying the core where it was possible. The Cortex-M0 core itself can do hardware multiplication but cannot do division, and instead the compiler must use its own routines which are much slower. The STM32F072 chip builds on that core and adds many useful peripherals. Mostly SPI and Timer were needed to communicate with the RF chip and to time the network. This chip has several internal oscillators and can be clocked without any external crystal with the disadvantage of the clocking imprecision. Additionally to the main crystal oscillator, it has 32.768 kHz crystal oscillator. This oscillator can be used to power battery backed Real Time Clock (RTC) unit or to calibrate one of the faster internal oscillators. The chip is not oriented for a low consumption. The 32.768 kHz oscillator is intended only to drive the RTC circuit, while main power is completely out, and not to wakeup the MCU from low power sleep. That summed with the unused components on the PCB means that the consumption is too much to measure anything useful.

Several more PCBs were made later in the thesis. The same PCB layout was used, but it was fitted with STM32L072 [6], [7] instead of STM32F072, and no unused component was fitted. That made the boards much better in the low-power domain. The 'L' in the name STM32L072 puts the chip into the low-power category. Both chips are almost the same from the outside, and they can be substituted one for the other, but the main differences are inside. The chip is based on Cortex-M0+ [8] core. This core is an evolution from Cortex-M0 in the direction of low-power applications, and most importantly it fixes mistakes made in the M0 core. For example, it gives back the Vector Table Offset Register (VTOR) which was missing in Cortex-M0 and allows the user to move interrupt vector table. The 'L' version of the chip has also modified peripherals. Some changes are simply fixes of old mistakes due to the chip being newer, but lot of the changes are oriented for optimized consumption.

When both MCU and the RF chip are put into their deepest sleep, the whole board consumes only 1.4 µA at 3.3 V of power supply. Even the FLASH memory of the STM32L072 was shut down to achieve this consumption. Details of the operation are described in the chapter 9.1.1.

There is a schematics of the fitted parts of the prototype PCB at the illustration 2. It is only a small portion of the otherwise complex PCB which is a creation of professionals at Jablotron Alarms. After the simplification, the board is powered by the connector JP1 previously used only for manufacturing purposes. The BOOT0 pin on this connector was shorted externally to the GND in the 'L' version of the board. This pin selects which memory is booted at startup. The BOOT0 pin on STM32L072 unfortunately lacks a proper pull-down resistor probably as a part of low-power optimizations. This meant that the chip was falling randomly into the bootloader and was not running the wanted code in FLASH. By pulling the pin down either by a resistor, or in this case by a short, this problem is fixed. The remaining pins, TX and RX, are connected to General Purpose Input-Output (GPIO) pins, or the pins can be switched to Universal Asynchronous Receiver/Transmitter (UART) peripheral. Both options are useful for the development and debugging.

The MCU is programmed and debugged through JP2 which is a standard connector for Serial Wire Debug (SWD). SWD is a two wire connection (SWDIO and SWCLK) commonly used to debug Cortex-M chips. The debugging was done by LPC-Link 2 configured as CMSIS-DAP. LPC-Link 2 is a hardware tool intended for chips of a company competitive to ST. CMSIS-DAP is an ARM standard for debugging which is hopefully gaining on importance. It is an unusual combination, but I don't own an ST-Link, and furthermore, from personal experience, the CMSIS-DAP run on LPC-Link 2 is much faster than the original ST debugger.

Apart from MCU and RF chip, there are only two buttons and two Light-emitting Diodes (LEDs). They shouldn't consume any power when they are not used.

Connection to the RF chip is through an SPI interface. MCU is the master. It selects the RF chip (putting the CS pin low) and transmits data on the MOSI pin while clocking on SCK pin. Concurrently, the RF chip sends responses on the MISO pin. The other signals connected to the RF chip are GPIO_0 and GPIO_1. The GPIO_0 signal is on its own pin, and GPIO_1 is hidden in MISO when CS is high. They can be both configured to various signals from the RF chip.

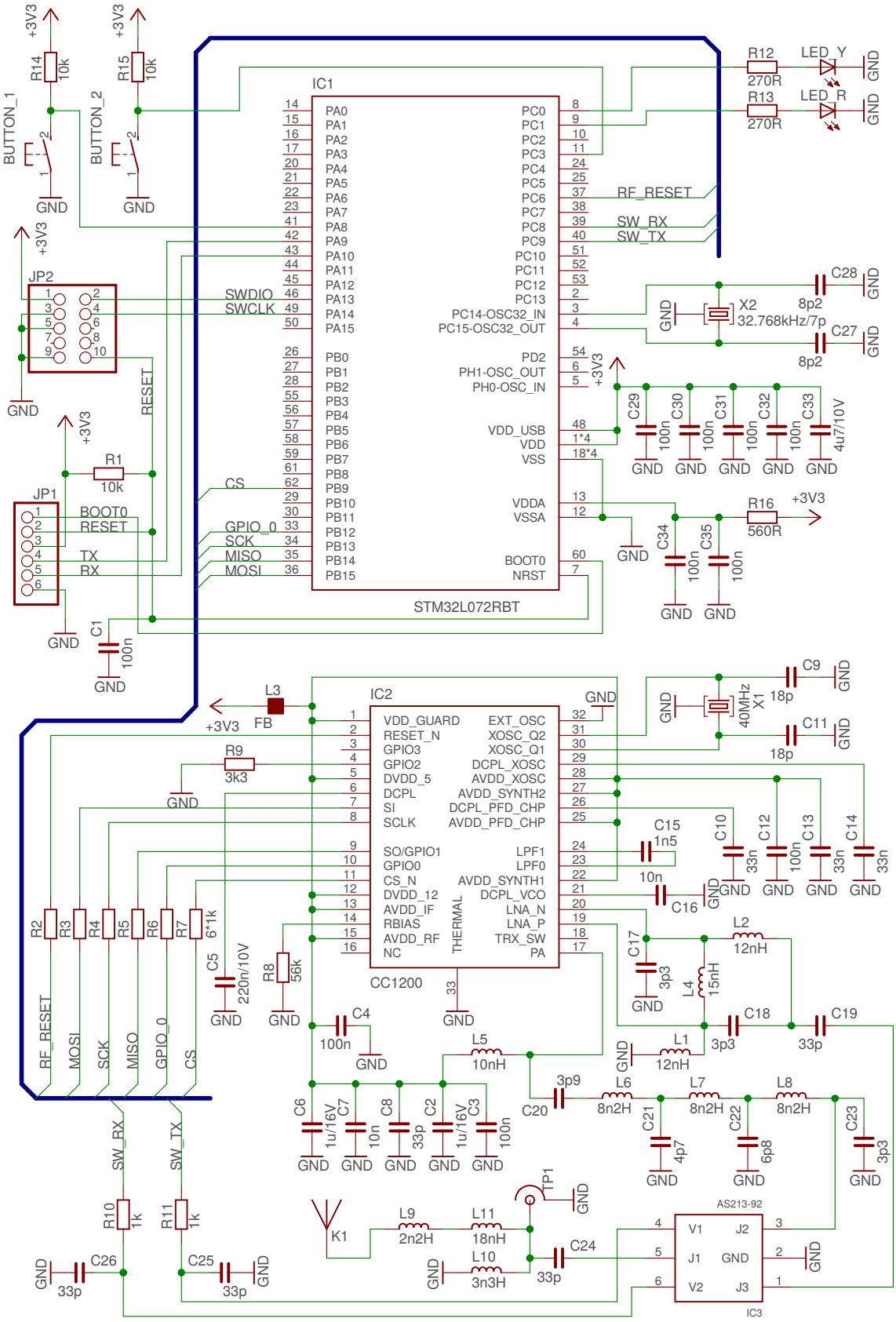

Illustration 2: Schematics, cut-out of used parts

3 Physical Layer

Radio equipment in the European Union is controlled by a Radio Equipment Directive 2014/53/EU [9], known as RED. It replaced the Radio and Telecommunication Terminal Equipment directive, known as R&TTE, which is now obsolete. The name R&TTE is still in use meaning the new directive.

The RED doesn't actually set any frequency bands. It instead appoints the Frequency Information System (EFIS) of the European Communications Office (ECO). This information system publishes lists of frequency bands, licensing and rules.

Devices using the unlicensed or Industrial, Scientific, Medical (ISM) frequency bands are called Short Range Devices (SRD) in the European Union. SRD frequency bands are described in an ERC recommendation 70-03 [10]. ERC is a historical abbreviation. The document is created by an Electronic Communications Committee (ECC) which is a part of European Conference of Postal and Telecommunications Administrations (CEPT).

The frequency band h1.1 is given in the assignment of the thesis. This band belongs to the category of non-specific SRDs. The ERC 70-03 states for the frequency band h1.1:

Band	-	863-870 MHz
Power	-	25 mW e.r.p.
Access	-	≤ 0.1 % duty cycle or Listen Before Talk (LBT)
Bandwidth	-	≤ 100 kHz for 47 or more channels (100 kHz preferred)
Intended for	-	FHSS

The frequency band is further specified by a norm EN 300 220 [11], [12], [13]. This norm is being revised in the time of writing of this thesis, and the new version V3.1.0 will soon replace the old version V2.4.1 [14].

3.1 Modulation

The modulation Gaussian-FSK (GFSK) is given together with the band and the RF chip in the thesis assignment. Simple Frequency Shift Keying (FSK) means a symbol is transmitted by either lowering or lifting the carrier frequency. There are just two symbols in this simple case, 1 and 0, and the bitrate equals to the symbolrate. The Gaussian preposition means a Gaussian curve changes the frequency instead of blunt rectangular signal. That helps narrowing the channel bandwidth.

The transmission power is limited by 25 mW e.r.p. which is an equivalent of +14 dBm.

$$10\times\log\left(\frac{25\,\mathrm{mW}}{1\,\mathrm{mW}}\right)=+13.979\,\mathrm{dBm} \qquad \textit{(Eq. 1)}$$

The vendor suggests a symbolrate of 38.4 kBaud and 20 kHz deviation with the input filter of 100 kHz. We can approximately check the bandwidth BW by the Carson's rule on the equation 2.

$$BW = 2 \times (BT \times sr + f_{dev}) = 2 \times (0.5 \times 38.4\,\text{kBaud} + 20\,\text{kHz}) = 78.4\,\text{kHz} \qquad \textit{(Eq. 2)}$$

BT is Gaussian filter bandwidth bit period product, sr is symbolrate and f_{dev} is frequency deviation

3.2 Channels

I selected 64 channels spaced by the recommended 100 kHz for this network. The power of 2 should simplify programming and speed the calculations later on.

The norm EN 300 220, in its part 2 [12], describes FHSS equipment. It states that FHSS shall not be used in the frequency bands for LDC/HR devices. These devices are described in part 3-2 [13] of the norm. There are four designated LDC/HR frequency bands marked A through D. Four channels that would overlap these frequency bands are skipped. The resulting FHSS bandwidth, including the skipped bands, ranges from 863.2 MHz to 867 MHz.

Table 2: Channel frequencies

Channel number	Center frequency
63	869.950 MHz
62	869.850 MHz
61	869.750 MHz
Reserved Band D	869.650 – 869.700 MHz
60	869.550 MHz
59	869.450 MHz
Reserved band C	869.300 – 869.400 MHz
Reserved band B	869.250 – 869.300 MHz
58	869.150 MHz
57	869.050 MHz
56	868.950 MHz
55	868.850 MHz
54	868.750 MHz
Reserved band A	868.600 – 868.700 MHz
53	868.550 MHz
52	868.450 MHz
- - -	
1	863.350 MHz
0	863.250 MHz

3.3 Clear Channel Assessment

The norm EN 300 220, in its part 2 [12], mentions that equipment using FHSS on the given frequency range is required to comply to the duty cycle limit of 0.1 % or shall use Listen Before Transmit, sometimes known as Listen Before Talk (LBT). Because of the beacons used in this network architecture, the duty cycle limit cannot be satisfied. The first part of the norm [11] describes the LBT to be Clear Channel Assessment (CCA).

The CCA method means that prior to transmitting the device must listen on the same channel. If the power received is lower than a given CCA threshold, the channel is clear and transmission can begin. The minimal time to receive is called the minimum CCA interval which is 160 μs [11].

The minimum CCA interval is an important difference from the old version of the norm. The old version didn't know the term Clear Channel Assessment and defined Listen Before Talk minimum listening time to be 5 ms + random part between 0 and another 5 ms. This period would make any attempts on FHSS with higher duty cycle futile. This limit probably considered the past radio equipment which couldn't make fast RX to TX turnarounds. The maximum deadtime or the time of the turnaround is defined as 5 ms, but for example CC1200 can make turnaround in 43 μs [1].

In my network, the CCA rules are applied before all transmissions. Enough time before each transmission the radio hardware is set to the receiver mode. While the radio is receiving, whole packet is stored in its FIFO memory. A single command is issued on the given time of transmission, and the radio automatically switches from the RX mode to TX mode monitoring the channel to the last moment. An interrupt is issued by the radio if the transmission doesn't start. The same principle is used not only for the polite spectrum access but also for arbitration in the network which is described further in the thesis.

4 Existing FHSS Solutions

Now that the regulations have been introduced, is a good time to look for existing FHSS technologies. Actually, there is not much to look at. There are only a few proprietary solutions, and there is Bluetooth. There are for example muRata HyperMesh modules, TI's Dolphin modules or Neocortec modules. The first two are not available for the European 868 MHz band, and all three of them are using some kind of FHSS communication which is not publicly documented.

Last technology worth mentioning is the IEEE 802.11-1997 standard. It is the original 802.11 which is now marked as legacy and not used. It was based on FHSS or Direct Sequence Spread Spectrum (DSSS) with Carrier Sense Multiple Access and Collision Avoidance (CSMA/CA).

4.1 Bluetooth

Bluetooth [15] is an FHSS network oriented for a regular consumer and his Personal Area Network (PAN). It operates in the unlicensed 2.4 GHz band. The core specifications in their fourth version define several types of physical and link layers and a complicated system of higher layers.

The first physical layer is referred to as the Basic Rate (BR). It is based on GFSK with Bandwidth-Bit period product (BT) of 0.5 and frequency deviation of 115 kHz. The symbolrate is 1 Mbaud.

The second physical layer is Enhanced Data Rate (EDR). This improvement is backwards compatible. If the device wants to use an EDR, it starts transmitting in the BR. There is a guard interval after the header during which transmitter and receiver both switch to another modulation. The available modulations are π/4 rotated Differential encoded Quaternary Phase Shift Keying (π/4-DQPSK), for 2 Mbps, or Differential encoded 8-ary Phase Shift Keying (8DPSK), for 3 Mbps. The symbolrate stays for both modulations the same as for BR.

Both BR and EDR use 79 frequency channels starting at frequency 2402 MHz with 1 MHz for one channel. Both modulations use the same link management. All devices are synchronized to master's clock, and transmissions happen alternatively in 625 μs long slots, even slots for master and odd slots for slave. Connection of devices which know each other is done using the page scan channel. A paging device transmits short packets repeated in half of regular slot time with a different hopping pattern. A page scanning device listens with its own hopping pattern and responds.

The third physical layer is the Low Energy (LE), sometimes called BLE or Bluetooth Smart. This layer uses only 40 channels spaced by 2 MHz, starting at the same frequency 2402 MHz. It uses again the GFSK modulation with BT of 0.5, but it uses higher frequency deviation of 185 kHz. That is the reason for the 2 MHz channel bandwidth.

Three of the 40 channels are defined as advertising channels, and the other 37 are defined for the data. The symbolrate is again 1 Mbaud. Connection of two devices is started by one of them which is advertising on advertising channels by transmitting. The transmissions are done in pseudorandom intervals. After each transmission, the advertising device listens on the same channel. When the other connecting device, the scanning device, responses, they both switch to a connection state. When in connection, the master starts an event by transmitting in the predefined time window. Both devices then alternate in transmitting until the end of an event. The communication is not in a fixed time raster, as for BR and EDR, but is chosen dynamically.

The Bluetooth LE is oriented to the lowest possible power, but the advertising and scanning process takes a lot of time. According to a research paper [16], the connection process takes 1.15 s which is much above the requested response time. The announced 5th version of the Bluetooth core specifications will perhaps be more successful.

The fourth physical layer is Protocol Adaptation Layer (PAL). It is just an interface for the IEEE 802.11-2007 which hides all the suffixes: a, b, d, e, g, h, i and j.

4.2 Tyco PowerG

Tyco PowerG [17] is a proprietary technology aimed at domestic wireless alarm systems. The technology overview claims it to be FHSS, but in the 868-869 MHz frequency band it uses only 4 channels. EN 300 220-1 [11] states that FHSS must use at least 47 channels spread over a wider frequency band. It uses 50 channels in the American 912-918 MHz frequency band and could be called an FHSS technology. The limitation for the European SRD band is caused most likely by the restrictions given in Belarus, Ukraine and Russian Federation.

The frequency switching happens 64 times a second, and the slot length is 15.625 ms. The technology overview further claims that the sleeping devices stay synchronized into the network with accuracy of ±30 µs and consumption of less than 2 µA while periodically sniffing for an incoming message. 30 µs is a too good fit to be a coincidence. My guess is that it is a period of the 32.768 kHz clock crystal. It seems that this is a completely different approach than described in this thesis. Their devices appear to be fully synchronized and sniffing for incoming packets which are sent to wake the devices up. The crystal oscillator has to be running and calibrated to the network clock.

Common 32.768 kHz crystal can slow down by 0.035 ppm per °C. For an outdoor device, the temperature can change by 5 °C in no time. By the equations 3, 4 and 5 we can calculate how long does it take for such oscillator to lag behind by 30 µs.

$$T_{slow} = \frac{1}{(1-a)\times f} \qquad \textit{(Eq. 3)}$$

f is the crystal frequency, a is the frequency tolerance, T_{slow} is the period of the slower oscillator

$$n = \frac{T_{diff}}{T_{slow} - T} \qquad \textit{(Eq. 4)}$$

n is the number of ticks needed for a difference of one T_{diff}, T is the original period

$$t = n \times T_{slow} = \frac{T_{slow} \times T_{diff}}{T_{slow} - T} = \frac{\frac{T_{diff}}{(1-a) \times f}}{\frac{1}{(1-a) \times f} - \frac{1}{f}} = \frac{T_{diff}}{a} \approx 2.86\,min \qquad \textit{(Eq. 5)}$$

$a = 5 \times 0.035\,\text{ppm}$, $T_{diff} = 30\,\mu\text{s}$

The resulting time of 2.86 minutes is just approximate, but it still shows a problem with synchronized sleeping devices. If the synchronization broadcast would be 2 ms long and sent once every minute, it would mean receiving ratio of 33 ppm. That is 783 uA of cumulative current consumption for CC1200 while not counting sleeping current and periodical sniffs. Either the parameters are a bit exaggerated, or PowerG uses a lot better HW, or said devices on a cloudy day consume a lot more power because they are repeatedly heated by sun, cooled by wind and have to resynchronize with the network.

4.3 Non-FHSS competition, IEEE 802.15.4

IEEE 802.15.4 is a specification of the lower layers using either 2.4 GHz, 868 MHz or 915 MHz frequency bands. It uses mostly DSSS and only one channel at the given 868 MHz frequency band. This technology doesn't suffer from most problems in this thesis, but as DSSS it probably isn't as good as FHSS when it comes to a bad radio environment.

Most known implementation is the ZigBee standard which adds networking and higher layers. ZigBee is intended for networks of sensors powered by batteries or by energy harvesting. This would be another very promising candidate for a network with the given parameters.

5 Link Layer

In the link layer I will describe timing and synchronization of the network including when can which device transmit and when it should receive. As this thesis is oriented to fast response times, one of more important parts is a synchronization of a slave that was sleeping. Another part of this chapter is also the arbitration process as a very simple method of minimizing number of collisions.

5.1 Network timing

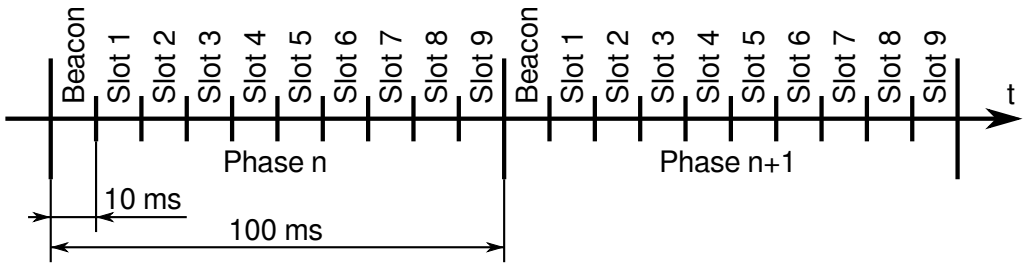

Illustration 3: Network timing

The time is divided into 2^{16} **phases**, each one 100 ms long, and one phase is divided into 10 **slots**, each 10 ms long. The current frequency channel is given by a phase, a slot and a **map** numbers. The map number omits some bad channels and selects four particular channels as the beacon channels.

5.2 Synchronization of the sleeping slave

As stated before, to conserve power the slave needs to sleep for a long time. When the slave wakes up, it must synchronize itself to the network and send a packet in 400 ms.

A sleeping slave remembers its map number, but because it's sleeping, it lost track of time. It cannot know what phase or slot numbers are valid right now. From the map number the slave knows the four selected beacon channels. It scans the beacon channels looking for a beacon packet after it wakes up.

CC1200 can check for a preamble of a packet in two ways. First is the Carrier Sense (CS) which is a bit faster but depends on the signal strength and is fairly unstable. The second method is the Preamble Quality Threshold (PQT) which looks for a matching preamble. This method is more stable, but it takes approximately 630 μs to complete one sniff.

The slave repeats these sniffs and alternates its frequency channels. Sooner or later it must randomly sniff the right channel where someone is transmitting a preamble. When the slave detects a preamble, it stays receiving. If the packet is a valid beacon from the

right master, it takes the time of the syncword, the phase number and the new map number. These three pieces of information are enough to be synchronized to the network.

The needed length of the preamble can be estimated by multiplying the number of channels which need to be sniffed by the length of one sniff on equation 6.

$$t_{\mathrm{PR}} \approx 4 \times 630\,\mu\mathrm{s} = 2.52\,\mathrm{ms} \qquad \textit{(Eq. 6)}$$

The time t_{PR} can be converted to the lengths of a byte on equation 7.

$$t_{\mathrm{PR}} \times \frac{br}{8} = 4 \times 630\,\mu\mathrm{s} \times \frac{38.4\,\mathrm{kBaud}}{8} = 12.096\,B \qquad \textit{(Eq. 7)}$$

The 12 byte preamble should be enough for the scanning slave to catch it.

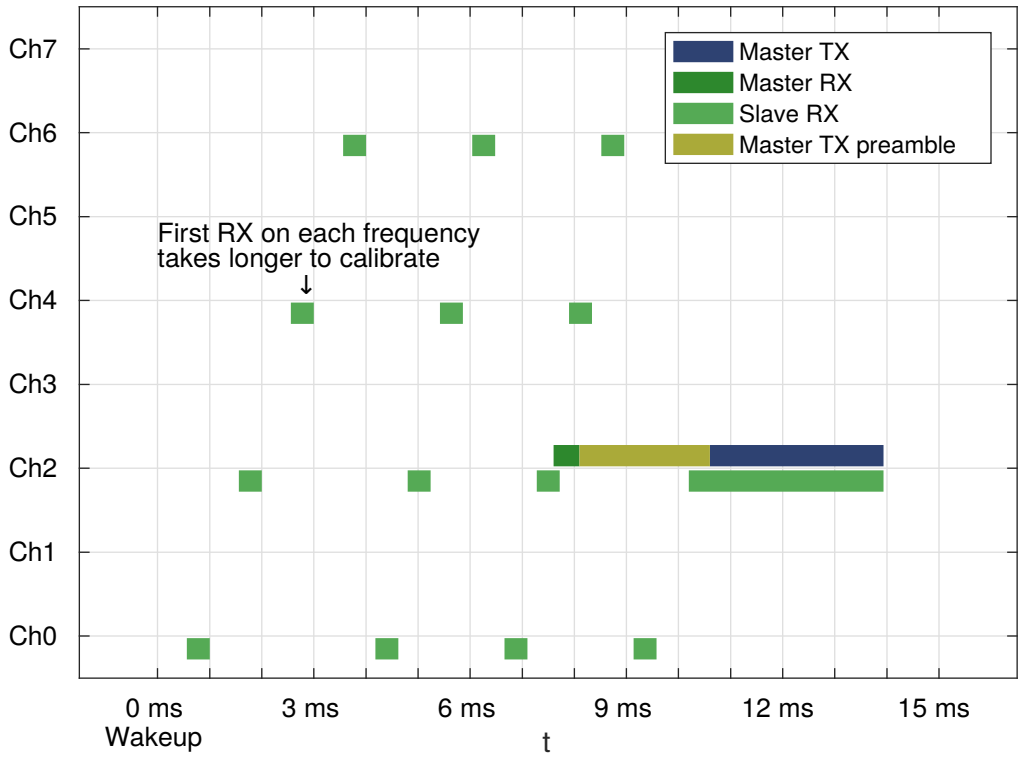

Illustration 4: Scanning for the beacon

The resulting process is depicted at the illustration 4. First sniff on each beacon channel takes a little bit longer because the RF chip needs to calibrate its internal regulators. The following sniffs are shorter because the MCU can directly write the necessary values obtained by a previous calibration into the right registers.

5.3 Timing of a slot

The phase and slot start with master's syncword. There is always a master's syncword for the beacon and a phase start. For the other slots, master may be transmitting in which case the slave can resynchronize its clock, or master is receiving which means that slaves must continue with their internal timing from master's most recent transmission.

The packet can take maximally 7 ms. The next 2 ms are reserved for oscillator imprecision, and the last 1 ms is for arbitration. When we consider 1% oscillators inside the MCUs then the receiver can be maximally 2 ms later than the transmitter after the 100 ms phase.

In conclusion the timing is as follows:

- From 0 ms to nearly 7 ms there can be a transmission.
- From the end of a packet to the 7 ms all devices must switch to the next channel.
- From 7 ms to 9 ms every device must be listening.
- The last millisecond is the arbitration and start of the next transmission.

The resulting timing can be seen on the illustration 5.

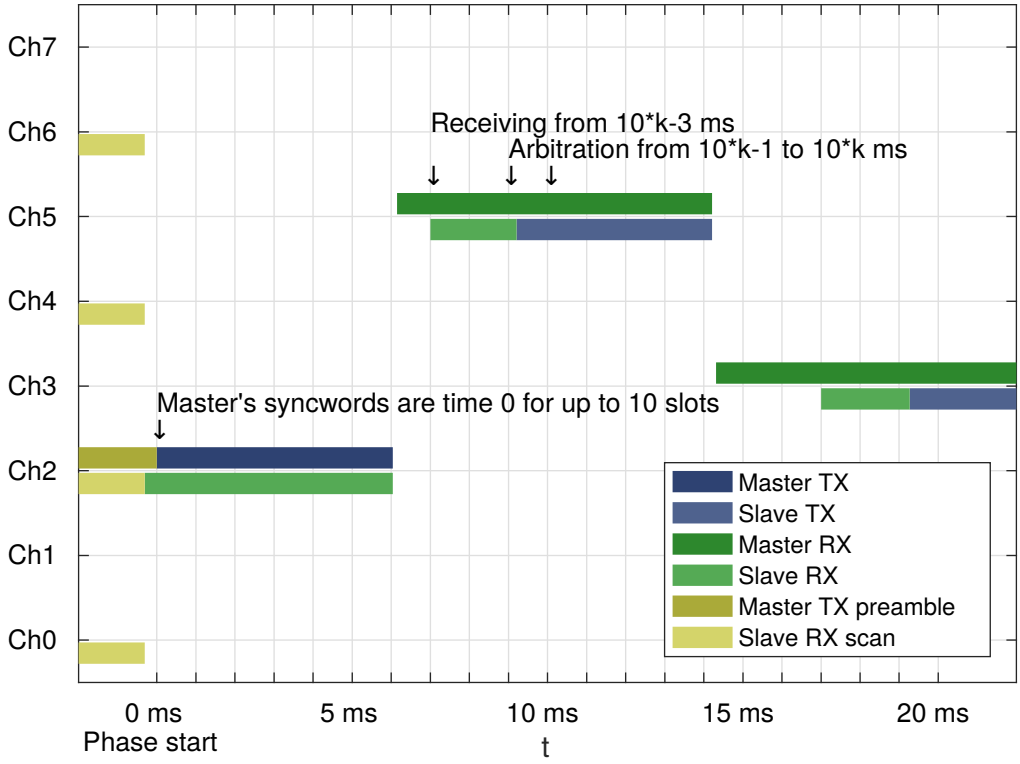

Illustration 5: Slot

Because of the longer preamble in beacon packets, the last slot must be finished maximally at 7th millisecond with no regard to the oscillator imprecisions. This can be solved by forbidding the last slot for slaves not precisely synchronized to the master and by shortening the longest possible packet in the last slot.

The arbitration is just a simple random delay maximally 1 ms long. Because there are slaves that are not synchronized in the network, no synchronized collision avoidance method can be used. This way all slaves select randomly a time between 0 and 1 ms with as fine resolution as they can. The randomness of this selection can be radically improved by using an internal random generator embedded inside the CC1200. The RF chip uses the radio aerial and ambient noise to further randomize the number. If the slave is lagging

behind the network, it is less likely to win the arbitration. However, the slave may measure period between two consecutive beacons, and it will be better synchronized the next phase and have better chances.

5.4 The phase

Zeroth slot in each phase is used as a beacon slot. Master uses longer preamble and transmits vital data to synchronization of the network. This beacon slot is always used by the master. In each master's packet there is an information whether the master is going to transmit in the next slot. This way the master doesn't have to participate in an arbitration, and its syncword can be used to synchronize in every slot and not just the beacon. When the master stops transmitting, slaves may take up the arbitration.

Slaves which don't want to transmit may sleep for the rest of the phase. Only the master must stay receiving. Once the slave looses track of packets, it must wait for the next phase. Reason are the longer packets described in the next chapter.

Out of 10 slots one is always beacon, 6 can be used by master and 3 are reserved for slave. Master doesn't use the last 3 slots. That gives slaves a chance to transmit even when master is transmitting as much as it can.

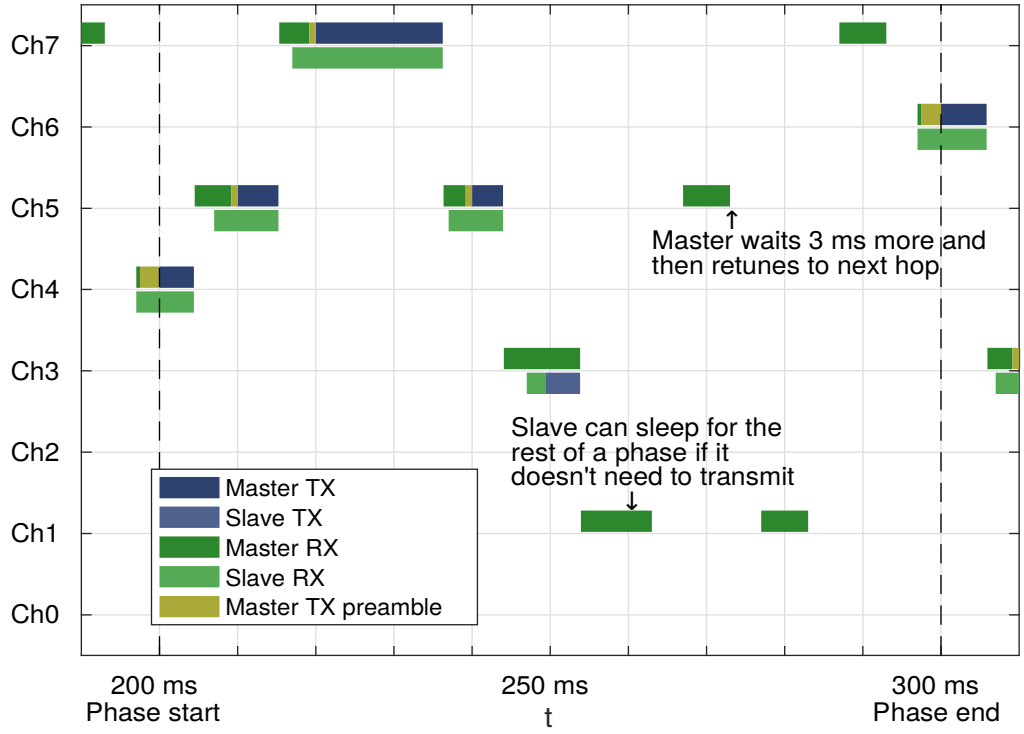

Illustration 6: Phase

Illustration 6 is a detail from illustration 1. Master sends a beacon on the channel 4 at time 200 ms. Slots 1, 2, 3 and 4 are also used by a master. Only slots from 5 to 9 can be used by slaves. The information that master stops transmitting is sent with the packet in slot 4.

5.5 Longer packets

A longer packet can be transmitted spanning more than one slot to save packet stuffing. This is similar to Bluetooth BR/EDR [15]. I have selected a limit of the packet length to 128 bytes (shortened to B further on), including the preamble and Cyclic Redundancy Check (CRC), to fit inside the First In First Out (FIFO) memory of the CC1200. That limits the packet to maximally 3 slots. During the packet transmission the channel stays the same, and the next hop frequency is omitted and unused. This can be used only if there are enough slots left. If the slot number is 5, master can send only a double packet because slot 7 is reserved for slaves. The same applies to slaves and the beacon slot.

Table 3: Longer packets

Slots taken	Data in the air	Duration in the air	Data for a higher layer
1	33 B	6.875 ms	12 B
2	81 B	16.875 ms	60 B
3	128 B	26.667 ms	64 B + 42 B

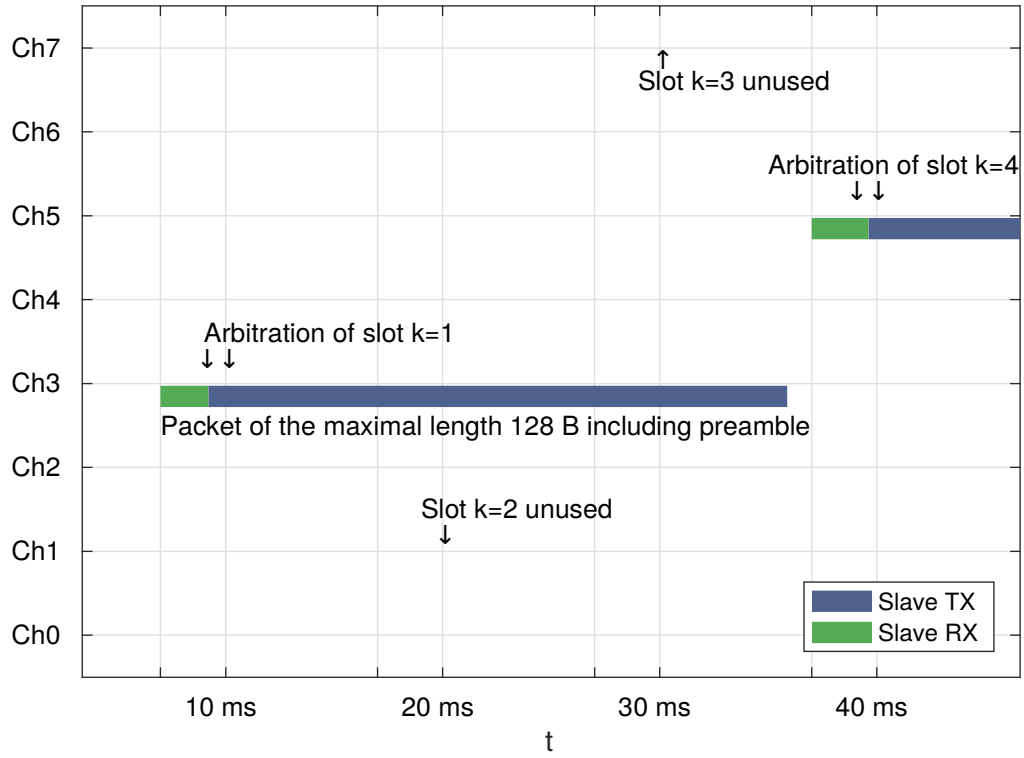

Illustration 7: A longer packet

The resulting 3-slot packet is depicted on the illustration 7. Slave starts transmitting in the regular arbitration interval but doesn't finish before 7th millisecond and continues. The slave must end before 27th millisecond to finish at the right time of the third slot. Other devices stay receiving to the longer packet and channels 1 and 7 are omitted.

6 Hopping Sequence

The ERC recommendation [10] gives several conditions. One of them is that in the given frequency range at least 47 channels need to be used. In the h1.1 band it is not necessary, but a good practice is to implement also the Adaptive Frequency Agility (AFA). That means changing the frequency bandwidth to fit the ambient conditions and other transmitting devices which use the same spectrum.

6.1 Used and eliminated channels

We have 64 possible channels from chapter 3.2. We need to eliminate some of them because of the multipath interference caused by the building where the system will be installed and narrowband interference caused by other wireless technologies. Together with four beacon channels, everything must be encoded into a single 16 bit map number for the fast synchronization.

The process when the map number is changed can be simplified if the map number is split into two independent halves, further referred to as **halfmaps**. One halfmap stays the same while the master changes the other halfmap. Slaves connected to the system still have half the information and know 2 beacon channels for which they can listen. Each halfmap controls half of the spectrum, respective 32 channels. The lower halfmap controls channels 0 to 31, and the upper halfmap controls channels 32 to 63. One bit of information can be saved if the higher halfmap stores the same numbers 0 to 31, and 32 is added during the calculations.

The 8 bit halfmap is split into three parts: size of the eliminated area, position of the first eliminated channel and a random number in range from 0 to 3. The group can be 1, 2, 4 or 8 channels wide. The highest bit that is zero sets the size of the eliminated area:

- `0b0eeeeerr` to eliminate 1 channel
- `0b10eeeerr` to eliminate 2 channels (e bits are position of the eliminated area)
- `0b110eeerr` to eliminate 4 channels (r bits are random)
- `0b111eeerr` to eliminate 8 channels

The two lowest bits are always random. When the conditions would force two systems into the same map, there is still a chance 3 in 4 to get a different halfmap and 15 in 16 to get a different map number.

A halfmap controls only 32 channels, and it is a 5 bit number in range from 0 to 31. Because there is not enough space, only the higher bits are stored in between the size bits and the random bits. This means following combinations:

- 1 eliminated channel can be any channel from 0 to 31
- 2 eliminated channels can start only on even numbers from 0 to 30
- 4 eliminated channels can start only on channels divisible by 4 from 0 to 28
- 8 eliminated channels can start only on channels divisible by 4 from 0 to 28

Furthermore, there is a combination where the size is 8 eliminated channels and the starting channel is 28, respective halfmap = `0b111111xx`. The first 4 and the last 4 channels are eliminated from one half of all channels in such case.

The 8 bit halfmap must also encode two beacon channels, and those channels cannot be in the eliminated area. I've picked a solution where the beacon channels are a fixed distance, 10 channels, from the eliminated area. Added to that is the random part of the halfmap, ranging from 0 to 3. The beacon channels are a modulo of division by 32 to limit the beacons into the right area of 32 channels. The beacons are interlaced, so the lower halfmap sets beacons 0 and 2, and the higher halfmap sets beacons 1 and 3. The beacon channels are calculated as follows:

$$beacon = (e - 10 + r) \bmod 32 \qquad \textit{(Eq. 8)}$$

e *is first eliminated and* r *is the random part,* $\bmod$ *is the modulo operation*

respective:

$$beacon = (e + s + 10 + r) \bmod 32 \qquad \textit{(Eq. 9)}$$

s *is the size of the eliminated area*

6.2 Adaptive frequency agility

The simplicity of the map number limits the network from any kind of sophisticated frequency agility. Only one group can be eliminated in each half of 32 channels. The obvious choice is to start with the worst channel.

There are two possible indicators to determine which channel is the worst. First is the Link Quality Indicator (LQI) measured by the CC1200 chip. It has no physical unit, but out of two values compared together the higher number means worse link. The second indicator is number of failed LBTs or collisions. If the number of collisions is higher than 10 then the channel with the most collisions will be the worst channel. Otherwise, the worst is a channel with the highest LQI.

The size of the eliminated group is always decided by the LQI of the channels in the group. Only one group of each size contains the worst channel except for the group of 8. I neglect the possibility of 8 eliminated channels starting on a number not divisible by 8 for simplicity. The conditions for the size are:

- If the other channel from the group of 2 is at least ¾ as bad as the worst channel, eliminate 2 or more channels.
- If the other 3 channels from the group of 4 are on average at least ¾ as bad as the worst channel, eliminate 4 or more channels.
- If the other 7 channels from the group of 8 are on average at least ¾ as bad as the worst channel, eliminate all 8 channels.

The process of changing the map number takes a lot of time. Master must wake up all the slaves, get the LQI statistics from them and inform them when is a good time to wakeup for a new map. Because of the sleeping slaves, it must take at least 20 minutes.

6.3 Synchronized sequences

The selection of a frequency used for communication is a pseudorandom sequence. There is a lot of methods to get sequence that seems random to the outside world but is predictable from both ends of the communication. I needed a method that is not too sequential. The method must give the next channel number by a single calculation from the phase and map numbers obtained in a beacon.

Methods using shift registers, such as gold sequences, would be hard to implement on processors as the used Cortex-M0+. I started with, perhaps the simplest and oldest method of generating pseudorandom sequences, the Linear Congruential Generator (LCG). One step of the generator are just two operations, multiplication and addition. The truncating is done automatically by the 32 bit wide overflowing maths used in the processor. I have selected the generator from the Numerical Recipes [18] which is truncated to 32 bits and uses multiplier a = 1372383749 and increment c = 1289706101. When the generator is reset with a seed value, the value is just stored in the memory. Each new pseudorandom number is then:

$$new = \left(\left(a \times last \right) \bmod 2^{32} + c \right) \bmod 2^{32} \qquad \textit{(Eq. 10)}$$

To synchronize the generators in both ends of the communication I reset the generators on each beacon with the information present in the beacon. The seed is composed of the phase number as the lower 16 bits, the map number as the higher 16 bits and XORed with the master's physical address. There could be two systems with the same map, but there can never be two systems with the same master's physical address.

The generator creates a sequence of 9 numbers 32 bit wide. The channel number is obtained by XORing the generator output down into a 6 bit number, and creating something in the area of combined generators with XORshift being successor to LCG. If the selection should find a channel that is in the eliminated area or one of the beacons, the selection repeats. The resulting channel is a composition of different bits of the random generator output best explained with the C code:

```
channel = (rng ^ (rng >> 10) ^ (rng >> 26)) & 0x3f;
```
(Eq. 11)

`rng` *is the 32 bit generator output,* `channel` *is the number of the channel selected*

6.4 Simulations

The following illustrations were made by simulating channel selections of a system. Unfortunately, standard autocorrelation or crosscorrelation, which are used for binary sequences, cannot be used here. The channel number is a nominal variable, and it doesn't matter if one system selects channel 2, and the other selects channel 3 or channel 63. Important is only when the channels match because that leads to a collision.

Two channel selection sequences were simulated to investigate how much would two independent systems collide. One of them was shifted in time while counting the number of matching channels. This created a sort of 'cross-comparison'.

One channel selection sequence was calculated to investigate hidden periodicities. It was shifted in time and compared to itself. This created a sort of 'auto-comparison'.

All illustrations are simulated for 16 random map numbers and physical addresses. The maximum of all samples is shown if not stated otherwise. This better describes the worst case scenario because one random simulation could be just a lucky combination. The sequences were calculated for all existing phases which means 10×2^{16} selections.

6.4.1 Channel number from LCG output

I selected XORing three parts of the LCG generator output together instead of just using the 6 lowest bits. This is in agreement with the chapter 7 Random Numbers of the Numerical Recipes [18]. The authors proclaim, in this book, that LCG generators are outdated, and lower bits of LCG are not random at all. They suggest using several types of combined generators. One simpler type for 32 bit arithmetic is XORshift generator being a successor of an LCG. This means that the result of LCG is put as a seed to the XORshift generator, and only its result is used. My implementation is not truly a XORshift generator, but the principles seem to work similarly.

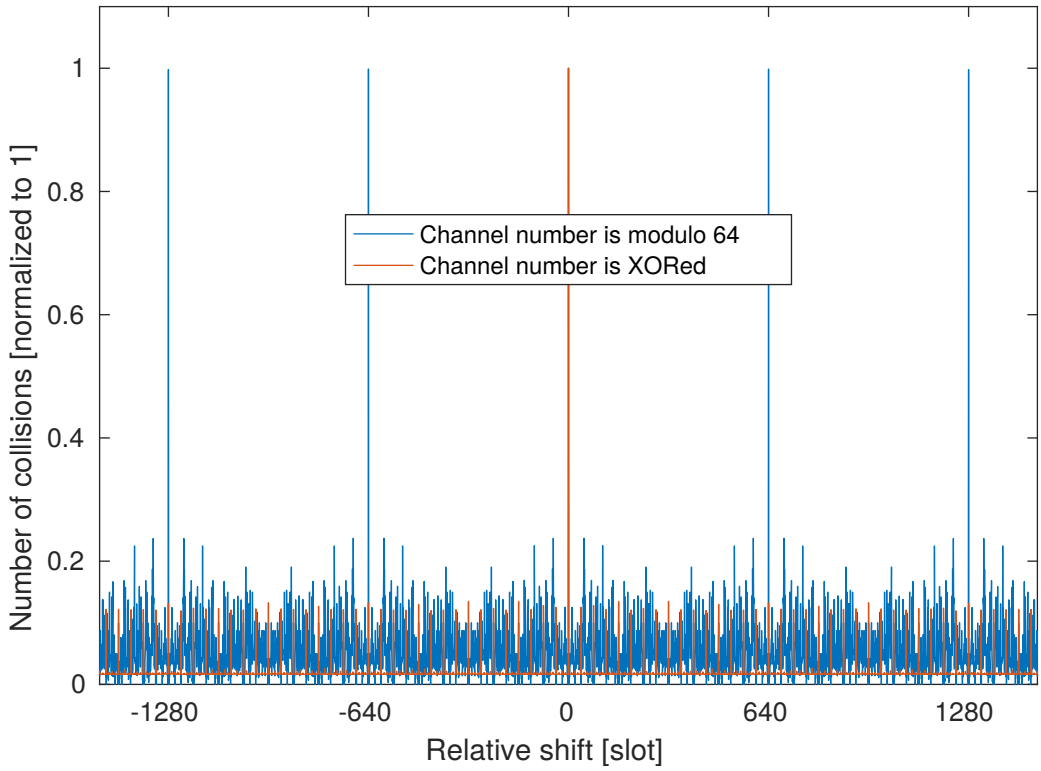

Illustration 8: Channel number from generator output (auto-comparison)

The illustration 8 shows that when the channel number is just a modulo of the LCG output then the result is highly periodic after 640 slots which is 64 resets of the generator. When the channel number is a XORed composition, as described in chapter 6.3, the only periodicity is when the shift is a multiple of 40 slots when the beacons repeat.

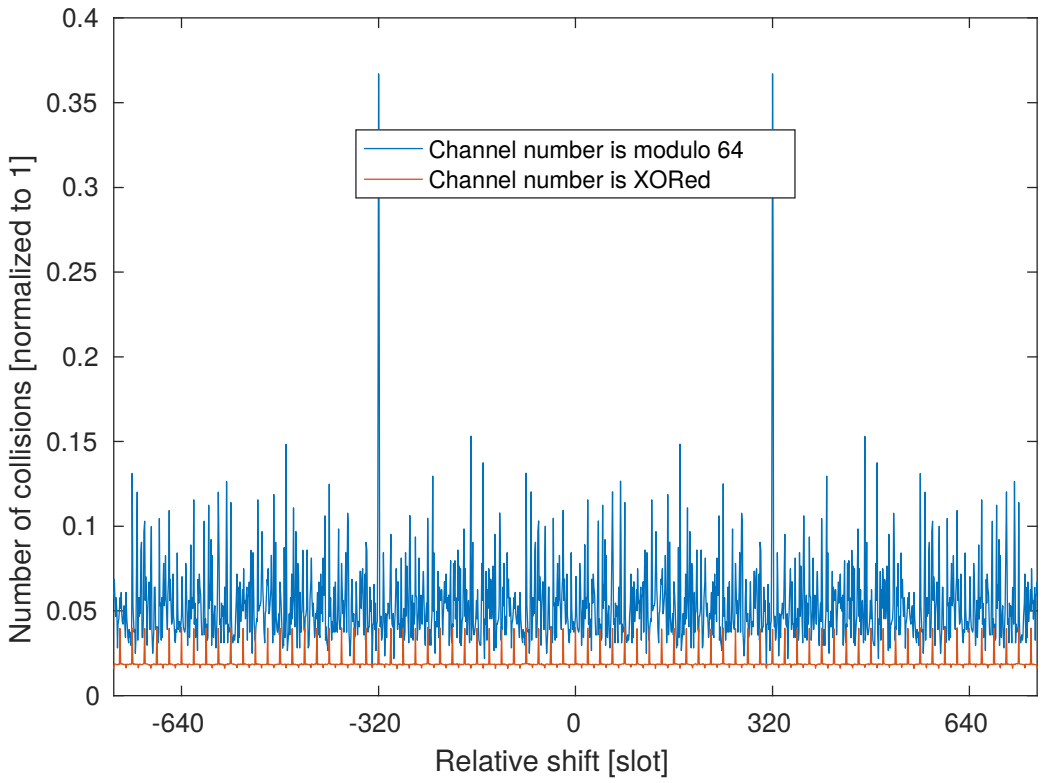

Illustration 9: Channel number from generator output (cross-comparison)

The similar properties can be seen when comparing two independent systems on illustration 9. When only the modulo is used, then the collisions are much more common and more erratic.

6.4.2 The expected collisions

We can estimate collisions of two independent systems as if the selection was purely random and compare that to the simulation. Truly random selection of two systems would collide in exactly $1/64 \approx 0.0156$ of all selections. In this system, the beacons and between 2 to 16 channels are removed from the selection pool, and the selection is used only in 9 out of 10 slots. That means a collision would ideally happen between $0.9 \times (1/58) \approx 0.0155$ and $0.9 \times (1/44) \approx 0.0205$ of the time.

The beacons themselves make the most influencing part. If one beacon matches, then there is a collision once every 40 slots when the beacons repeat. The overall chance of collision is increased by $1/40 = 0.025$ of the time. After the summation, the peaks when the beacons match should be approximately 0.0406, 0.0656, 0.0906 or 0.1156 high. This corresponds to the simulation on the illustration 10.

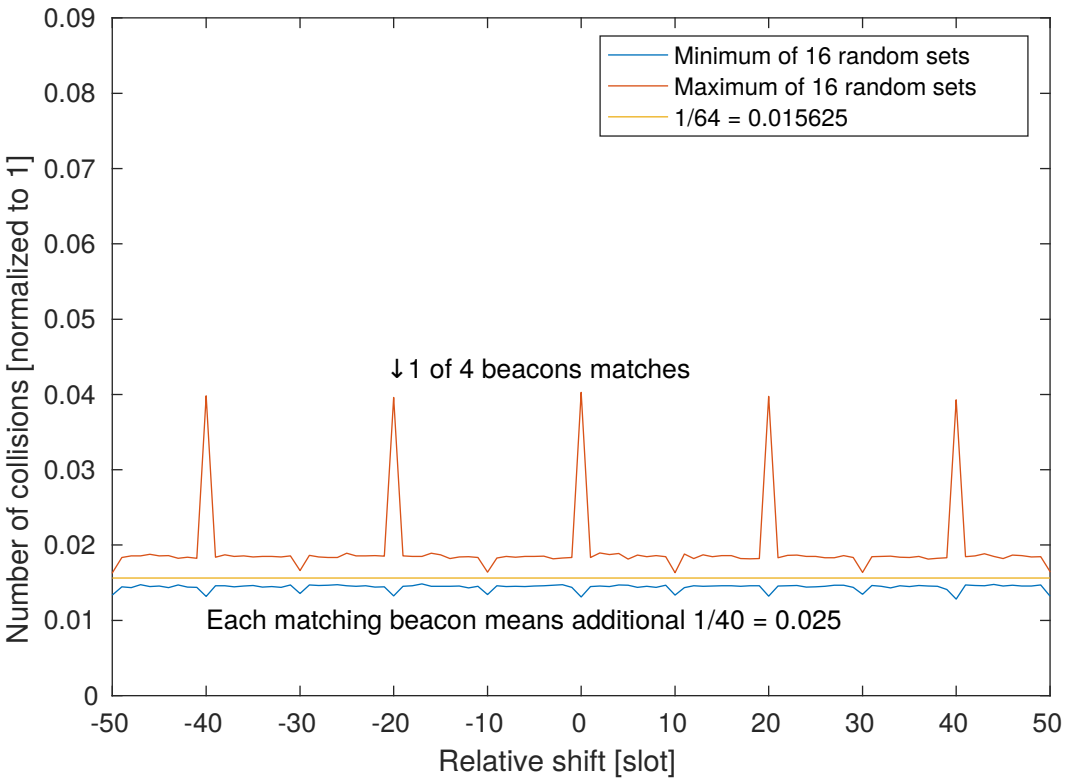

Illustration 10: Expected collisions of two systems (cross-comparison)

Collisions on beacons would be fatal for the system, but there is no way of eliminating them if I want to preserve the fast synchronization. The two systems would have to choose the same beacon channel and the right time to get the collision. I guess that the oscillator imprecisions would slowly shift the two systems apart, and the collisions would happen only for a short time and then disappear for a longer time period.

7 The Packet

Each device has its own 4 B address. The address of each prototype is a composition from 12 B MCU manufacturer's identification. This simplifies the development, as the same firmware has different addresses on different boards. The addresses of production devices will have to be given inside the firmware.

All numbers following in chapters 7 and 8 are stored little-endian. This means that bytes of lower significance are stored first in the memory or are transmitted first on the air.

7.1 Packet

Table 4: The packet

4 or 12 B																				
Preamble	Syncword				Length	Master address				State	Payload								CRC	

The **Preamble** is 4 B for any packet and extended to 12 B for the beacon.

As a **Syncword**, I have selected a value `0xa1a581f2`. The syncword is used by the RF chip to detect end of the preamble and the beginning of the packet and to ignore packets of other systems. There is a possibility that some other developer may choose the same physical radio parameters and the same syncword, but in real world that possibility is neglected.

Length byte tells the RF chip how many bytes are following while not counting 2 B of CRC. The RF chip needs this value to automatically evaluate the packet, do whitening and calculate CRC.

Master address is used to identify one wireless cell managed by the master. Slaves use the same master's address in this place as a destination.

If the highest bit of **State** byte is set, the packet is from master to slave. Reset value means that the packet was transmitted by a slave. The second highest bit, **More** bit, is used only in direction from master to slave, and it tells slave if master is going to transmit the next slot. The rest of the state byte is reserved for unexpected needs of the link management and other layers.

The **Payload** area are the useful data of the packet. There can be more than one payloads in the payload area. The slave address is encoded in each payload, so master can send one packet with multiple payloads to different slaves.

Two **CRC** bytes are automatically added and checked by the RF chip. From the available options, I selected CRC16 ($X^{16} + X^{15} + X^2 + 1$) initialized to `0xffff`. If the CRC field doesn't fit, the whole packet is thrown away.

The packet is whitened to eliminate the DC component of the data. The RF chip offers XORing with a PN9 sequence ($X^9 + X^4 + 1$) initialized to `0x1ff`. No data are actually added, so there is the same possibility that the data will have strong DC component, but the usual combinations of `0x00` and `0xff` translate to something pseudorandom.

7.2 Payload

Table 5: The payload

Paylength			Slave address (optional)		Data						
LM or higher layer	Slave address follows	Length									

Payloads in the payload area are separated by the length byte, which I called **paylength.** The paylength byte contains two flags. The highest bit indicates whether the payload is for the Link Management (LM) or for a higher layer. The second highest bit indicates if there is a slave address following the paylength. The last component is length of the remaining data. The value is ranging from 0 to 63 and corresponds to 1 B to 64 B following the paylength or the slave address.

If the slave address is not present, it is taken from the previous payload. If the address is not present in the first payload, it is a broadcast address. Only master can send broadcasts. Slave packets have sender address in the first payload, and all following payloads have the slave address omitted.

8 Link Management

The link management takes care of login to the master, sleeping and frequency agility.

The names of the LM payloads are an enum type of language C called `lm_hdr_t`. The byte values of the enum are used in header of the payload stored on the first byte of the payload data.

8.1 Beacon

As mentioned earlier, the master sends regular beacons on four beacon channels. I have limited the beacon packets to only one slot to save power of sleeping slaves. First payload in this packet is always LM_HDR_BEACON.

Table 6: LM_HDR_BEACON

LM_HDR_BEACON	map		phase	

Map and **phase** numbers are for the channel selection for the frequency hopping and are described in chapters 5.4 and 6.3. Map number determines the channels, and phase number increments with each beacon, respective with each phase.

8.2 Login

There are three packets involved: LM_HDR_LOGIN, LM_HDR_LOGIN_ACK and LM_HDR_GET_OUT. All three have no parameters.

The slave that wants to log into the network scans for any beacon. This scanning must be done over all 64 channels because slave doesn't know the map number. That takes a little bit longer than scanning from sleep described in chapter 5.2. When the link layer synchronizes with the master, the slave asks by using LM_HDR_LOGIN. Master decides if the slave is wanted in this network and responds positively with LM_HDR_LOGIN_ACK or negatively with LM_HDR_GET_OUT. The slave can repeat the LM_HDR_LOGIN several times if it gets no answer.

In the first case, on the illustration 11 left, the slave considers itself logged in with this master and sends information about its sleep preferences.

In the second case, on the illustration 11 right, the slave stops its synchronization. Now it depends on the application layer whether the slave doesn't want any other master or if it looks for any available master. The latter case lets the slave remember this master and scan again for a different one where it can ask again.

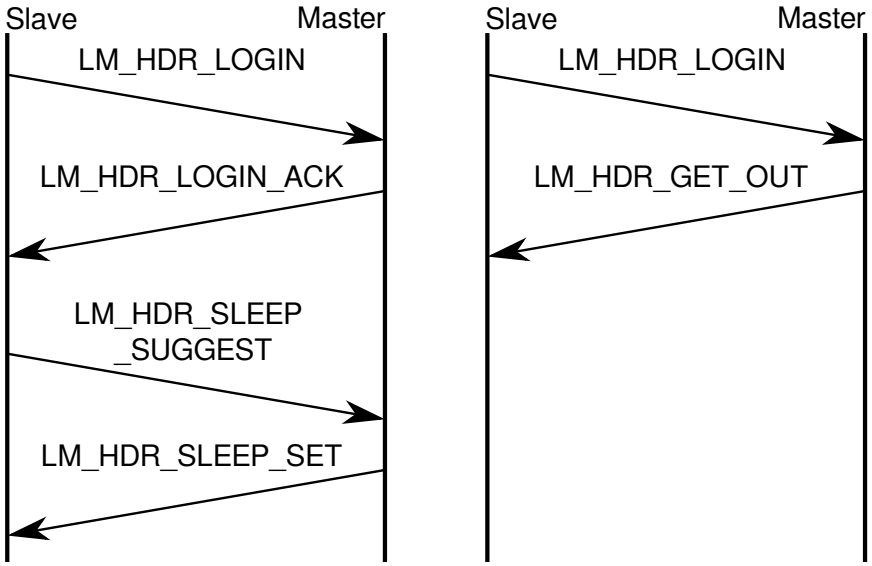

Illustration 11: Slave logging in

8.3 Slave going to sleep and returning from the sleep

The second step after login, on the illustration 11 left, is the information about sleep preferences LM_HDR_SLEEP_SUGGEST.

Table 7: LM_HDR_SLEEP_SUGGEST

LM_HDR_SLEEP_SUGGEST	skip	

The **skip** number suggests number of phases that the slave wants to skip between two beacons received when sleeping without loosing the synchronization. If the skip number is 1 then the slave doesn't want to sleep at all. If the skip number is 0 then the slave wants to sleep fully, sleep for a long time, loose its synchronization and use scanning after wakeup. The slave sleeping fully cannot be waken by master.

Master responds to this suggest by LM_HDR_SLEEP_SET.

Table 8: LM_HDR_SLEEP_SET

LM_HDR_SLEEP_SET	skip		alive	

The **skip** number in this payload is the number allowed to the slave. If master wants to keep the slave alive, it sends skip = 1, and the slave cannot go to sleep. This can be seen on the illustration 12 right. A slave allowed to go to sleep is on the same illustration on the left.

Alive is a phase number when the slave at latest must wake up and call the master to verify its presence. If the master wants to change the map number, it can tell all slaves to wakeup at the same time and check that all slaves successfully switched to the new map.

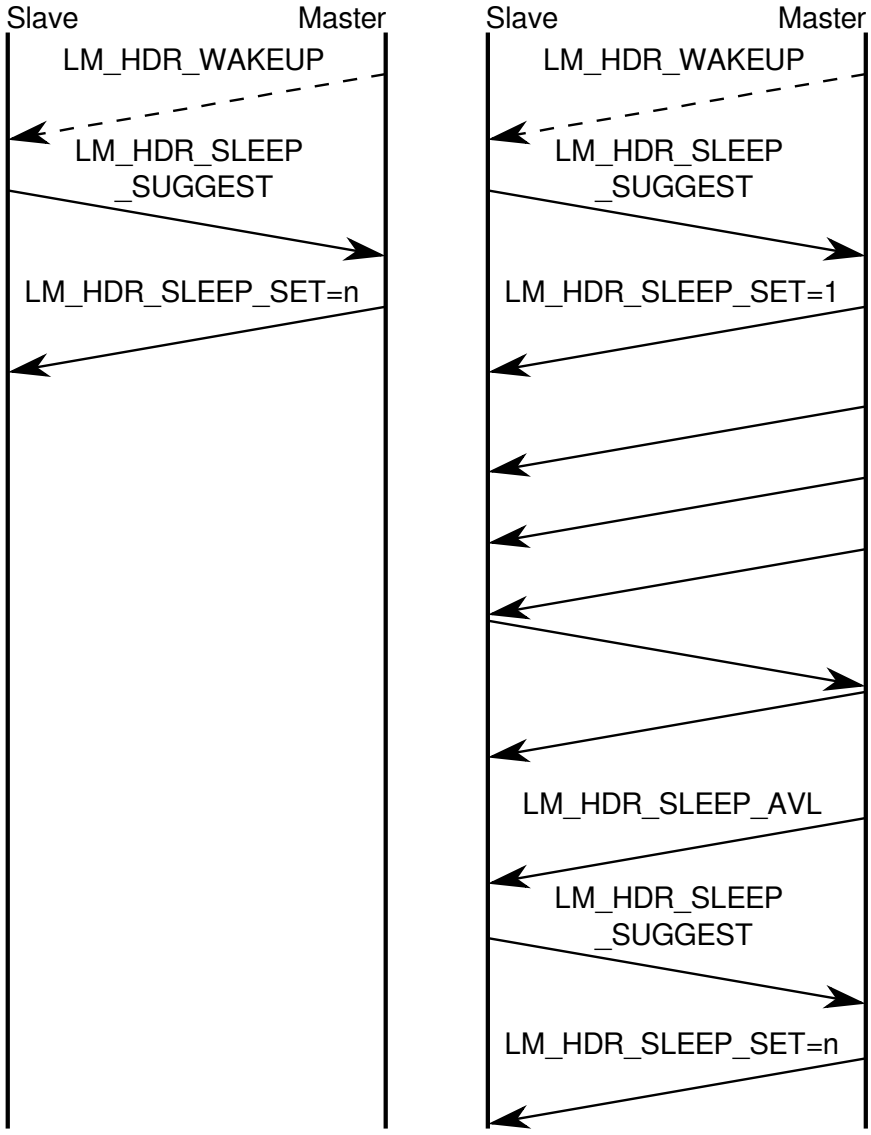

Illustration 12: Slave waking from and returning to sleep

Slave which would like to sleep but is not allowed needs to be informed when things change, and it is no longer needed. The header LM_HDR_SLEEP_AVL, without any parameters, does exactly that. Master repeats this packet until slave responds with LM_HDR_SLEEP_SUGGEST or until the slave is considered lost. After that, the master can send a slave to sleep using the regular LM_HDR_SLEEP_SET. This three packet exchange is needed to be sure that the slave went to sleep. Slave goes to sleep immediately after receiving LM_HDR_SLEEP_SET. The slave cannot verify that it went to sleep because if that packet would be lost, the master couldn't ask the sleeping slave

again. On the other hand, if the first packet got lost, the slave wouldn't go to sleep at all. The slave is surely informed about the availability of the sleep only by the use of three packets, and the slave makes sure that LM_HDR_SLEEP_SET answer is delivered. This exchange can be seen on the illustration 12 right.

Slaves that are sleeping and occasionally receiving beacons, with skip number higher than 1, can be waken from the sleep at will. Master appends one LM_HDR_WAKEUP payload, without parameters, right after LM_HDR_BEACON into the same packet. Unfortunately, only one such addressed payload will fit into this one-slot packet. This means that only one or all slaves can be waken at one time. The payload itself contains the slave address, so the LM_HDR_WAKEUP doesn't need any more parameters.

An example is a slave sleeping in 1 s intervals with skip number of 10. Such slave will receive a beacon and, disregarding the more bit, will return to sleep for another second. This happens even in a case when the master continues to transmit. As a part of the beacon packet, the LM_HDR_WAKEUP will tell the slave if it should wakeup, continue receiving the current phase and notify the master that it is alive.

8.4 Relogin

The LM_HDR_RELOGIN command could be used by several masters in cooperation. If one master measures weak signal, it tries to hand the slave over to another master in the same system composed of multiple smaller networks.

Table 9: LM_HDR_RELOGIN

LM_HDR_RELOGIN	master				map	

respective

LM_HDR_RELOGIN	master			

There is an address of the new master in the **master** field. The **map** number is provided, if known, to speed up the relogin process.

This function could be easily misused to steal slaves. A real production device would have to be locked by a higher encrypted layer. After the setup, the system would lock and wouldn't allow any structural changes.

8.5 Frequency agility

There are two headers needed for the frequency agility. One asks slave for the LQI statistics, LM_HDR_GET_STATS, which is without any parameters. The second header sends the statistics to the master, LM_HDR_STATS.

Table 10: LM_HDR_STATS

LM_HDR_STATS	lqi_0_0	lqi_0_1	...	...	...	lqi_m_n	...	...	...	lqi_7_5	lqi_7_6

The LQI is a 7 bit number for every one of 64 channels. If the values should be transmitted as 8 bit number, the number of bytes in the message would be 65. Payload limit is 64, so the 7 bit numbers must be squeezed together. Each **lqi_m_n** has an index $m \in \langle 0,7 \rangle$ and an index $n \in \langle 0,6 \rangle$. Every lqi_m_n stores LQI for a channel $8 \times m+n$ in the lower 7 bits. The remaining channels $8 \times m+7$ are split to 7 different highest bits of lqi_m_n with number of the bit being equal to n.

9 Software Architecture

All firmware for the MCUs was programmed using the C language. As Integrated Development Environment (IDE), I used the Eclipse C/C++ Development Tooling (Eclipse CDT) with System Workbench for STM32. The System Workbench for STM32, sometimes called AC6 or SW4STM32, is an Eclipse plugin developed by ST for development on their MCUs.

As a compiler, I used the GNU C Compiler (GCC) of type arm-none-eabi. The type means that the resulting code is cross-compiled for a bare metal ARM and not for any OS. For the debugger, I used the GNU Debugger (GDB) with Open On-Chip Debugger (OpenOCD). The first interprets the code and variables, and the second is for the connection with the hardware debugger.

The network firmware was developed as a static library of the Eclipse IDE. The whole library *MT-libnetwork* has three build configurations for each of the used boards. Two versions are for STM32F0 boards, with minor differences in used pins, and the last is for the STM32L0 board. There are symbol defines in the library settings for the processor and the board. These defines are used in the code as preprocessor conditionals. The preprocessor switches parts of code where the differences between MCUs needed a different program. Headers from this library are included in main projects, and the three precompiled libraries are used by GCC Linker when linking the main projects.

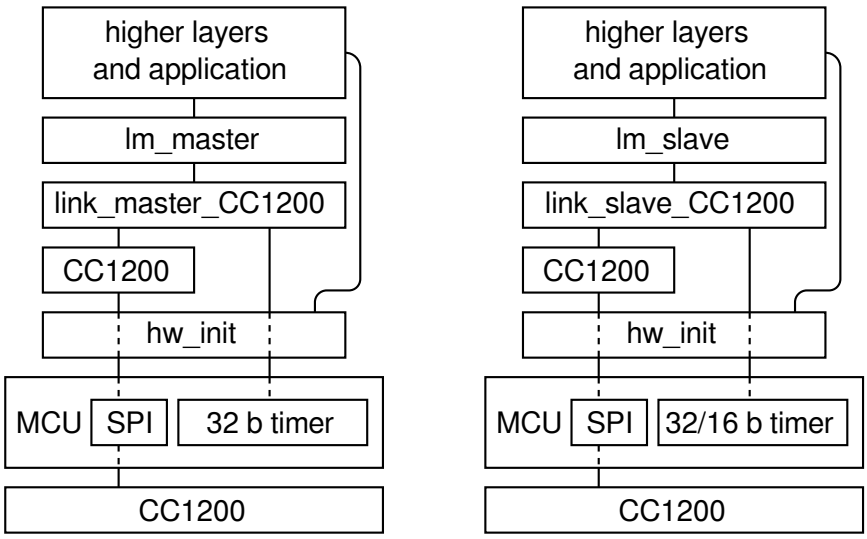

Illustration 13: Software layers

The library architecture can be summarized by the illustration 13. A similar structure was selected for both master and slave, and both are using the same low level drivers. A couple of source, '.c', and header, '.h', files typical for a C project will be called

a module in the following text. The header file is usually included into the source file of the module and into the main program where the module is used.

9.1 hw_init

This module contains functions and defines that select and control the basic hardware. It contains code for generally used peripherals but not code for timer or SPI used only by one of the higher layers. It initializes the hardware starting with the clock initialization. The clocks use an HSI internal oscillator and Phase Locked Loop (PLL) to multiply the frequency.

During the initialization, unused pins are set to a proper state. The right state for this family of MCUs is an analogue input. When a pin is in analogue input, the Schmitt triggers are turned off, and the chip consumes less power. Apart from the above, this module controls the PendSV interrupt, used for lowering the interrupt priority, SysTick timer or the EXTI controller of external interrupts.

9.1.1 Full sleep

The full sleep function has any relevance only for the STM32L0 version which consumes a lot less power. The Cortex-M processors normally enter a sleep mode using the Wait For Interrupt (WFI) instruction. This stops the clocking of the core, and the chip consumes less power until any peripheral throws an interrupt to the Nested Vectored Interrupt Controller (NVIC). All peripherals are kept clocked by the same system clock in this mode, and they consume power. The STM low-power family extends this by a STOP mode.

If the device enters the STOP mode, then all high-speed oscillators are stopped. The chip retains its RAM memory but can be waken only by a small number of peripherals one of which is the RTC. During the stop mode, the RTC can be clocked either by an external low-power crystal oscillator, LSE, or by an internal very inaccurate 37 kHz oscillator, LSI. Unfortunately, the RTC peripheral keeps the time in the Binary Coded Digital (BCD) format. It was a good and easy way a few decades ago, but now everyone must recalculate from regular integer time into a stupid BCD format and back.

The FLASH controller of the chip can be turned off to further reduce the consumption. This prolongs the wakeup time and requires code that puts the processor to sleep to be running from RAM memory. For a Cortex-M chip that is not a problem because it has both FLASH and RAM memories in one address space. It doesn't matter if a function is in RAM or if a variable is in FLASH. At least while the code doesn't attempt to rewrite the variable.

The linker distinguishes between *text*, *data* and *bss* memory sections. *Text* is a section entirely in the FLASH memory, and it stores functions and constants. The *bss* section contains variables and is only in RAM memory. After start, there are random data in the *bss* section. The last section, *data*, is in both FLASH and RAM. The processor copies this

section from FLASH to RAM during startup and creates variable but initialized memory. There is a section *ramfunc* in the linker file which is a subsection of *data*. By the compiler attribute `section("ramfunc")`, we tell the linker that this one selected function belongs to the section *data*. The function is then automatically copied into the RAM memory on start of the MCU, and other code calls directly its RAM address.

9.2 CC1200

This module controls the CC1200 chip through the SPI peripheral and DMA controller. It provides basic control functions and initializes all the registers, but the rest is on the higher layers. For example to get a received packet, an external interrupt goes right into the higher layer. The higher layer uses this module's reading and writing functions to decide what was the cause of the interrupt and to retrieve the data. Advantage of this is that the higher layer can read the packet by parts and process them right away. This requires less memory than if the packet was all retrieved by this module and offered whole to the higher layer.

Apart from SPI, this module also uses two DMA channels to get the data from SPI to RAM and vice versa. The core waits in the time the burst read or write is done, but the data are handled quickly, and the SPI frames can follow each other without any unnecessary delay.

9.3 link_master_CC1200

This module handles the master's part of the link layer. The main job is the network timing and control of the RF chip. This module is slightly dependent on the given MCU and highly dependent on the given RF chip. If any of them is changed, this library must be rewritten.

The base time is given by a 32 bit timer and its 3 match interrupts. The timer ticks every µs and is reset on match with the *autoreload register* at 99999 which is a period of one phase. All three interrupts are reconfigured each slot. The first interrupt is set to a time of slot start -3 ms. It changes the channel, starts receiving and, if the master wants to transmit, it also puts a packet into the RF chip's FIFO. The second interrupt starts the transmission at slot start minus preamble length. The third interrupt, at slot start +3 ms, ends the slot if the transmission or receiving didn't start. It also reconfigures the interrupts and prepares the variables for the next slot.

This module also contains two buffers for TX and for RX. Both buffers are payload oriented meaning that slave addresses and payloads are decoded and stored separately. The buffer is a barrel buffer with an additional barrel array of pointers to beginnings of each payload.

Lastly, this module is responsible for the channel selection and for the changing of half of the map number.

9.4 link_slave_CC1200

Part of this module is the same as in link_master_CC1200. The channel selection code is practically identical and the payload buffers are similar. Exception is that the address of the master is filtered and not stored in memory because the slave can talk only directly with the master.

The timing of this module is much more difficult. For one, the slave has states which radically change its behaviour. First and second states are for scanning for a system with given map or for any system. The difference is that a slave waking from sleep scans only four channels and remembers the RF calibration values to speed the scanning process. In the other state a slave is scanning all 64 channels, and because it doesn't need to find a system in a short time, it lets the chip to recalibrate on its own. The most important state is the *synced* state. The slave holds synchronized to the master in this state and can communicate.

All the timing is done using a floating base. A slave measures the length between master's beacons and adds one tenth to the interrupt match register each slot. The overflowing maths of the processor takes care of the rest. In the *synced* state the slave uses similar interrupts as a master. One interrupt starts receiving, second starts transmitting and the third ends the slot on specific conditions.

It turned out during some measurements that the *synced* state of this module is the most problematic part. The slave's half of this layer should have been built differently than master's half to better exploit specifics of the processor. This module has problems with short sleep and with resending blocked packets. If a master is blocked during LBT, it sends the packet again. The slave's part showed some unexpected behaviour, and that function was commented out. The other problem is that STM32L0 has a low-power timer, LPTIM, which can run from a 32.768 kHz clock crystal and can wakeup the processor from low-power modes. Since this timer has only one match register, a different approach would be necessary. A system of 'todo table' timing could work better and consume less power.

Because the processor STM32L0, unlike STM32F0, doesn't have any 32 bit timers, the time base was changed to 2 µs, and the time defines were halved. This was later in the development, so the module was split to a regular version and a 16bit version which is selected with a condition of the preprocessor.

9.5 lm_master

This module is the link management layer. It sends and receives payloads with the LM bit set to one. This layer holds the information about all slaves, puts them to sleep or wakes them up. One of its tasks is also to gather the information about channel quality statistics and plan the changing of the map.

This module should be platform independent. If the interface is written well, then it could be used on top of a different *link_master_xxxx* layer.

9.6 lm_slave

This module is a counterpart of the master's link management. This layer is split into several states similarly to the slave's link layer. First state is *login* where the slave asks master whether it may be connected to the network. From the first state the slave switches to *alive* state where it receives all the messages and can transmit data. The last state is *sleep* where the layer listens only for beacon packets and waits for wakeup payload hidden in them.

9.7 Size

The size of a simple slave's firmware without any compiler optimizations is 23428 B or 22.9 kiB and the size of a simple master's firmware is 23956 B or 23.4 kiB.

Table 11: Command arm-none-eabi-size output

Filename	*text*	*data*	*bss*
MT-slave1.elf	23416 B	12 B	3840 B
MT-master1.elf	23944 B	12 B	4688 B

The resulting firmwares are relatively large for just the lower layers of the communication interface. It would be on the limit of 24 kiB large sector in case of a bootloader. On the other way, the processors have both 128 kiB large FLASH memory, so the communication takes relatively only an 18 % of the memory.

An interesting thing to note is that the *data* section is only 12 B large. This means that the *size* command doesn't count functions in the *ramfunc* section.

10 Measurements

10.1 Response time

Response time from slave to master is one of the key parameters of the network. An external event on the slave must be received by the master in 400 ms.

An ordinary oscilloscope connected to the PC via USB was used to measure the response time. Once there, a Matlab function was used to collect the data. The first channel of the oscilloscope was wired to a button of the slave which is active low. It was also used as a trigger. The second channel was wired to a GPIO signal of the master. The signal was set high when the message was received.

For each measured scope, two trigger times were found. A falling edge of the first channel is the time when the button was pressed. A rising edge of the second channel is the time when the master received the information. I selected half of the logic voltage, 1.65 V, as a trigger for both. Delay between the two is the measured response time. A sample measurement is seen on the illustration 14.

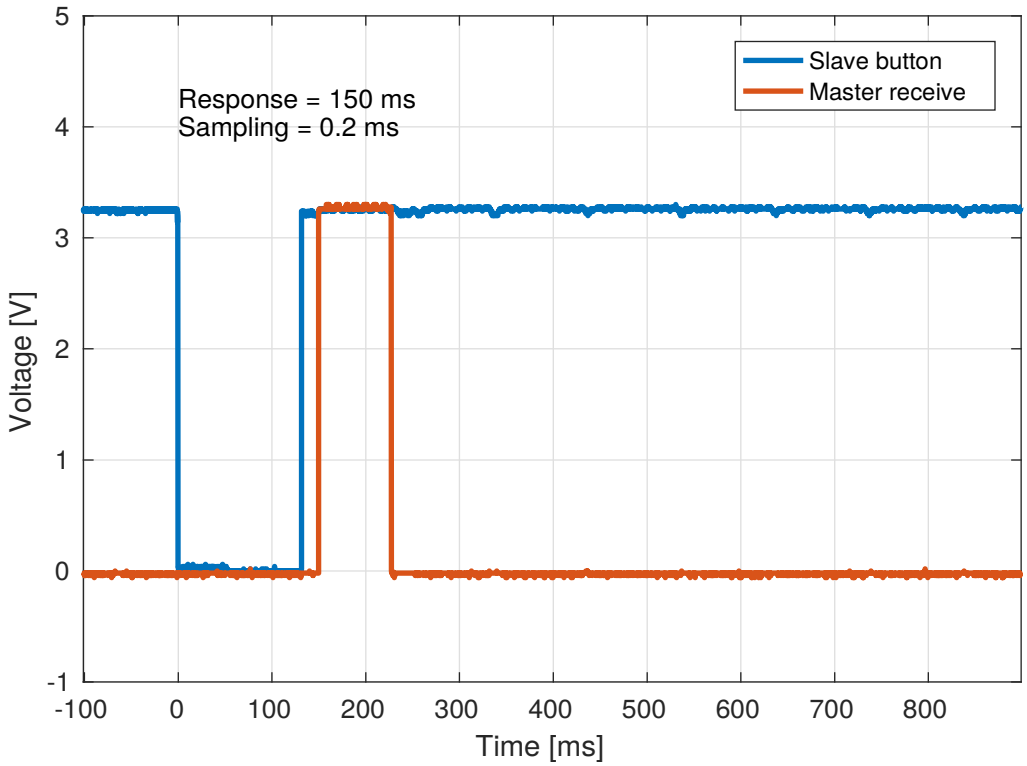

Illustration 14: Response time, sleeping slave to master

The measurement was done in relatively clean air, and no SRD device at the particular frequency range was near. This was verified only by a handheld spectrum analyser.

3 out of 100 measured cases were above the 400 ms limit which translates to 97 % reliability. Minimum, and the fastest response time achieved, was 51.6 ms. The mean was 135.2 ms with standard deviation of 90.2 ms.

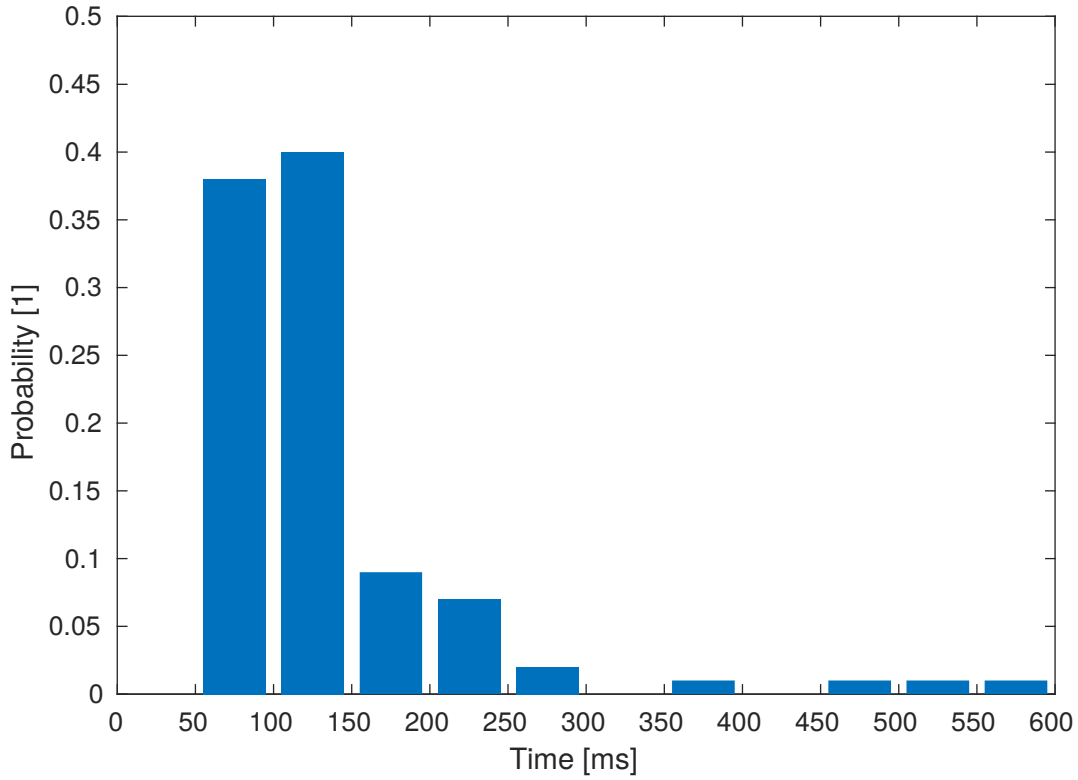

Illustration 15: Response time, sleeping slave to master, histogram

Histogram on the illustration 15 makes clear that in most cases the response time is around 100 ms. The other question is how this histogram would change if the measurements were taken in a noisy environment and on longer distances between transceivers.

10.2 Waking up slave sleeping in 1 s intervals

This measurement was similar to the previous chapter. First channel of an oscilloscope was connected to a button of the master. Second channel was wired to a GPIO output of the slave and toggled when the slave received the information. The slave wakes up some time before that, but the actual wakeup time is not important from an application point of view and wasn't measured.

Same as before, two trigger times were found. A falling edge of the first channel is the time when the button was pressed. A rising or falling edge of the second channel is the time when the slave received the information. The difference is that this time a press of a button toggled the output instead of setting it. As slave doesn't have any solid time base, the time when the output would reset could be misleading. The trigger was the same 1.65 V.

No near SRD device was found also this time. Spectrum was checked again by using a handheld spectrum analyser.

Only 16 out of 100 responses were longer than 1 s. This translates to 84 % probability to send an information to the slave in shorter time than is its sleeping interval. The mean was 636.2 ms with standard deviation of 313.4 ms.

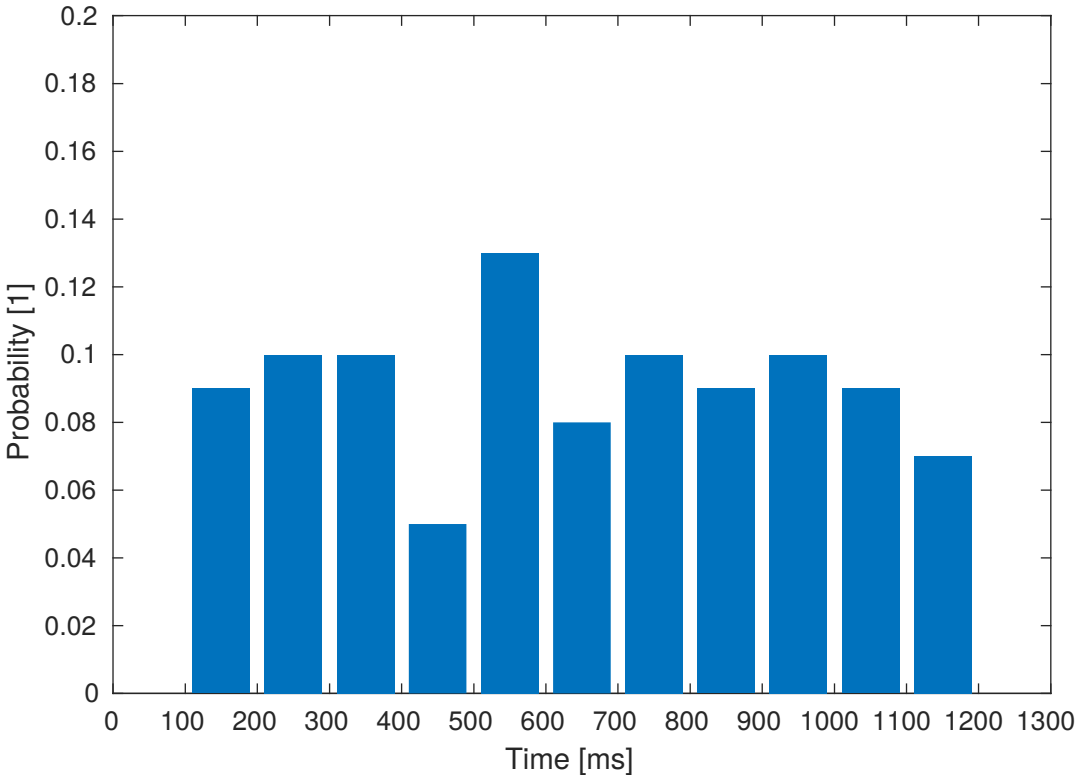

Illustration 16: Response time, master to 1 s slave, histogram

The histogram is displayed on the illustration 16. There we can see that the distribution is almost uniform, and the most important factor is the time when the button is pressed with regards to the time when the slave wakes up and listens for the beacon.

The process of delivering an information to the slave consists of three steps. First, the slave receives a beacon packet with wakeup command. Immediately after that, the slave notifies master that it is awake. In third step, master sends the information, but master can transmit again only after the next beacon. This additional 100 ms delay is manifested as an empty bin ranging from 0 s to 100 ms and non-empty bins from 1 s to 1200 ms.

10.3 Frequency spectrum

For the measurement of the frequency spectrum, I would like to thank the supervising company Jablotron Alarms and particularly Mr. Plecháč who did the measurement.

The measurement was done on a professional spectrum analyser with a direct connection to the measured board. The aerial was disconnected by unsoldering inductances which would split the signal into the analyser and into the air via the aerial.

Illustration 17 was done at frequency band from 866.4 MHz to 866.7 MHz with Resolution Bandwidth (RBW) of 1 kHz and Root-Mean-Square (RMS) detector. Illustrations 18 and 19 were done at frequency band from 863.1 MHz to 870.1 MHz with RBW of 3 kHz and again RMS detector.

All measurements were done with the hold function. The hold function is necessary to see all the channels used by the FHSS device and to see the short packets with long intermissions. Disadvantage is that the total power is greater than the momentary power of the transmitter.

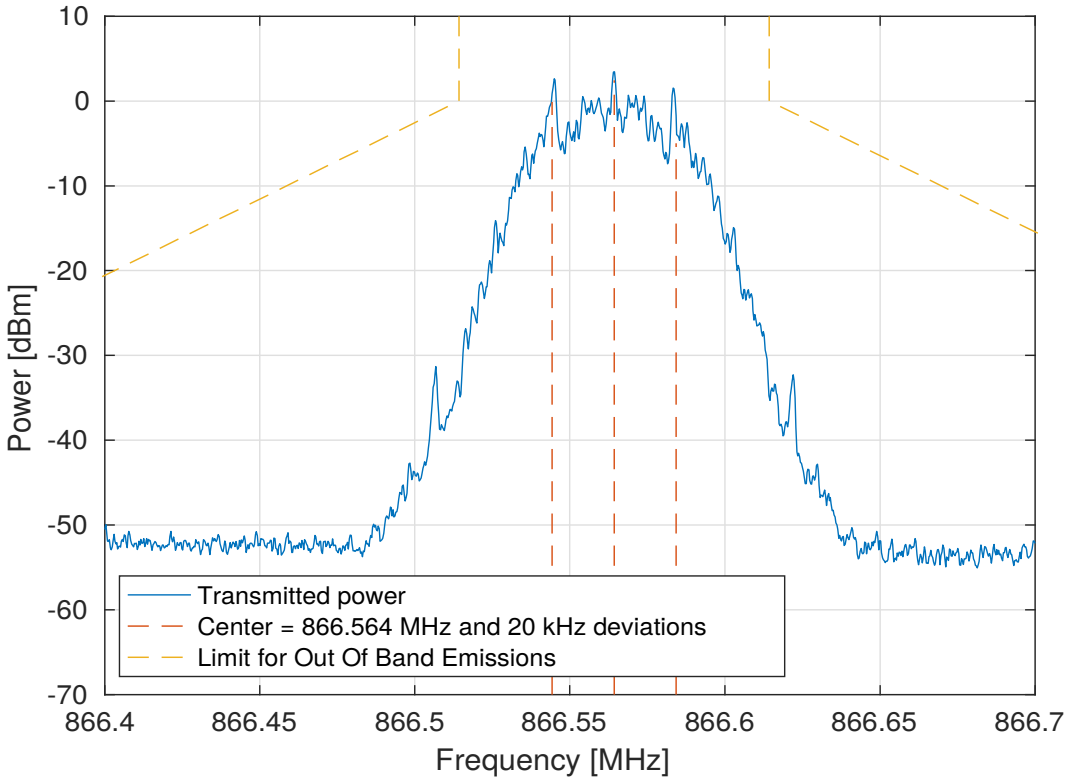

Illustration 17: Spectrum of channel 33

On the Illustration 17, there is a closeup of one of the beacon channels. The centre frequency of the channel 33 should be 866.55 MHz, but in reality it is 866.564 MHz. As this is a prototype, it was never properly factory calibrated. I will take the measured centre frequency for the discussion. There are two peaks at 20 kHz deviation from the centre frequency which represent logical bits being transmitted by the FSK modulation.

The EN 300 220-1 [11] specifies limits for Out Of Band (OOB) emissions. This means that the power outside the channel frequency band must be below specified limits. At the edges of the channel, the limit is 0 dBm per 1 kHz and declines towards -36 dBm per 1 kHz at the distance of 2.5 times the channel bandwidth from the centre frequency. The RBW for the measurement was exactly 1 kHz, so the limit applies as is. The limit is shown on the Illustration 17 and is clearly satisfied.

Another limit specified in [11] is the occupied bandwidth. It is defined as a frequency band where 99 % of the total mean power is located. The occupied bandwidth must be less than 100 kHz which is used as a channel spacing. Using the equation 12 I converted

$$A_{watt} = 1\,mW \times 10^{\frac{A_{dBm}}{10}} \qquad \textit{(Eq. 12)}$$

the measured data points to watt units. I summed all measured data points and summed all data points lying inside the shifted frequency band of 866.514 MHz to 866.614 MHz. The division on the equation 13 shows that the bandwidth condition is satisfied, and the estimate from the chapter 3.1 wasn't wrong.

$$\frac{\sum_{f=866.514\,\mathrm{MHz}}^{866.614\,\mathrm{MHz}} 1\,mW \times 10^{\frac{A_{dBm}(f)}{10}}}{\sum_{f=866.4\,\mathrm{MHz}}^{866.7\,\mathrm{MHz}} 1\,mW \times 10^{\frac{A_{dBm}(f)}{10}}} = 0.99981 \qquad \textit{(Eq. 13)}$$

$A_{dBm}(f)$ *are data points of transmitted power from Illustration 17*

The total power on the Illustration 17 was approximately +20 dBm or 4 times larger than allowed +14 dBm, but since measurement was done with the hold function, and that is expected. The total power is limited by a construction of the transceiver chip.

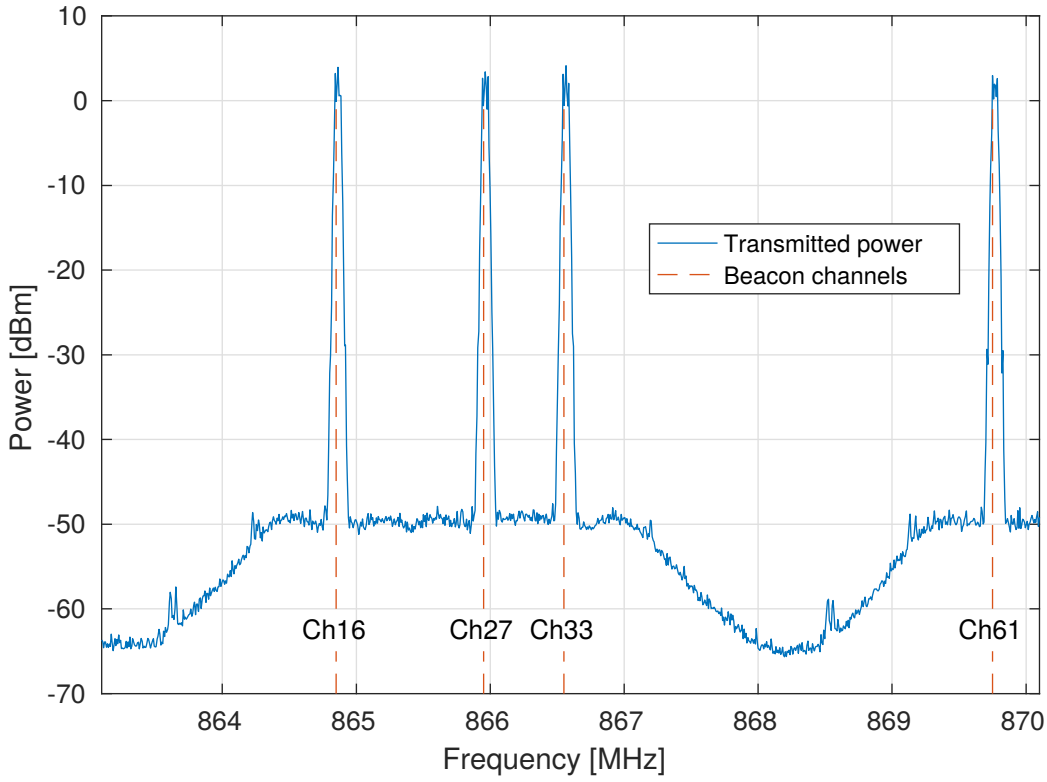

Illustration 18: Spectrum of idle system

There is a whole spectrum of the system on the Illustration 18. The system is idle and transmits only the beacon packets.

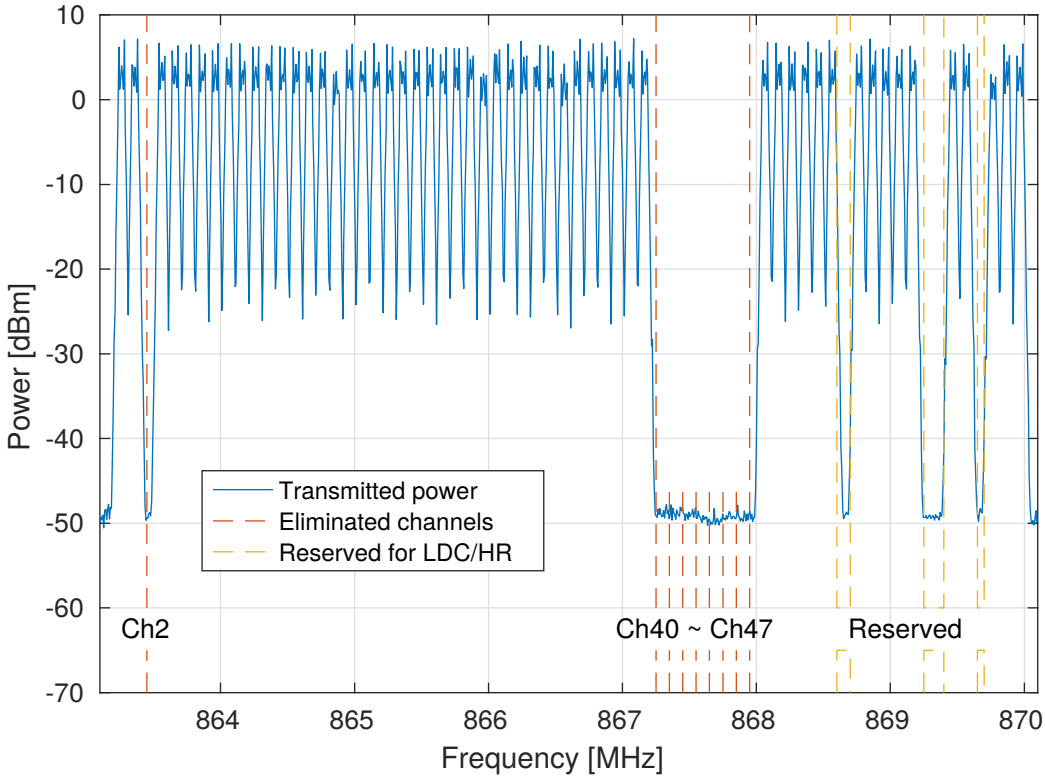

Illustration 19: Spectrum of busy system

On the Illustration 19, there is a spectrum of a system where master transmits in all slots in which it can. The gaps in the spectrum are the eliminated and reserved channels. We see that the frequency bands reserved for the LDC/HR devices are really clear on the right side of the illustration. The first gap from the left is the first eliminated group. Only the channel 2 is missing. The second and biggest gap is the second eliminated group. It consist of 8 eliminated channels starting at the channel 40.

10.4 Estimating consumption for a non-hopping system

To get a valid comparison, I shall first calculate estimates for the cumulative consumption of slaves in both frequency hopping and non-hopping solutions. If these estimates are valid and similar to the measured value, the comparison will apply even for a different MCU or a different RF chip.

The consumption of the MCU and the wide range of power down consumption of CC1200 is compensated by a value of 1.4 μA. I take this value as a base consumption in sleep and add it to the calculated RX and TX cumulated consumption. As the sleep current is very small compared to the TX current, I shall simplify the calculations by adding the sleeping current for the whole time even when the radio is in fact transmitting.

For the general idea, the consumption values written in the CC1200 datasheet [1] are:

- TX, +14 dBm 46 mA
- TX, +10 dBm 36 mA
- TX, +10 dBm, low-power mode 33.6 mA
- RX 23.5 mA
- RX, low-power mode 19 mA
- Idle 1.5 mA
- Power down, typically 0.12 µA
- Power down, maximally 1 µA

What exactly 'low-power mode' is, is unclear. The configurational software turns off an oscillator buffer for PLL and changes several reserved registers which are undocumented entirely. Presumably, the low-power mode is somehow worse.

10.4.1 Slave not receiving any data

A missing slave must be discovered under 20 minutes. This would mean transmitting once every 19 minutes and waiting for an acknowledge from the master. I approximate the lengths of packets by packets in this system. I take the *I'm OK* packet to be 24 B long, including preamble, and *Acknowledged* packet to be 26 B long. Receiving before the *Acknowledged* packet starts is neglected. This would mean TX time to be 5 ms repeated every 19 minutes. This divides to a ratio of 4.39 ppm.

$$t_{\mathrm{TX\_19min}} = \frac{24 \times 8}{38.4\,\mathrm{kBaud}} = 5\,\mathrm{ms} \qquad \textit{(Eq. 14)}$$

$$r_{TX} = \frac{t_{\mathrm{TX\_19min}}}{19 \times 60\,\mathrm{s}} \approx 4.39\,\mathrm{ppm} \qquad \textit{(Eq. 15)}$$

Similarly the RX time would be 5.42 ms and the ratio 4.75 ppm.

$$t_{\mathrm{RX\_19min}} = \frac{26 \times 8}{38.4\,\mathrm{kBaud}} \approx 5.42\,\mathrm{ms} \qquad \textit{(Eq. 16)}$$

$$r_{RX} = \frac{t_{\mathrm{RX\_19min}}}{19 \times 60\,\mathrm{s}} \approx 4.75\,\mathrm{ppm} \qquad \textit{(Eq. 17)}$$

Composed to a consumption it would be 1.71 µA.

$$I = r_{RX} \times 23.5\,\mathrm{mA} + r_{TX} \times 46\,\mathrm{mA} + 1.4\,\mu\mathrm{A} \approx 1.71\,\mu\mathrm{A} \qquad \textit{(Eq. 18)}$$

10.4.2 TX only Slave

The slave that only transmits must transmit multiple times to be sure that the information reached the master. For example, three consecutive packets every 9 minutes should be enough. The equation 19 gives transmitting 15 ms every 9 minutes.

$$t_{\mathrm{TX\_9min}} = \frac{3 \times 24 \times 8}{38.4\,\mathrm{kBaud}} = 15\,\mathrm{ms} \qquad \textit{(Eq. 19)}$$

It is 27.78 ppm in a form of a ratio from equation 20. The calculated predicted consumption could be approximately 2.68 µA.

$$r_{\mathrm{TX}} = \frac{t_{\mathrm{TX\_9min}}}{9 \times 60\,\mathrm{s}} \approx 27.78\,\mathrm{ppm} \qquad \textit{(Eq. 20)}$$

$$I = r_{\mathrm{TX}} \times 46\,\mathrm{mA} + 1.4\,\mu\mathrm{A} \approx 2.68\,\mu\mathrm{A} \qquad \textit{(Eq. 21)}$$

A completely different topic would be how much is the board cheaper if it doesn't contain receiver parts but only what is needed for transmitting, and if it has any impact on the consumption. This may be the reason why TX only devices are popular for purposes where there is no need to receive any data.

10.5 Estimating consumption for this system

10.5.1 Slave not receiving any data

The first stage after wakeup is scanning for a beacon. The scanning process may take 50 ms on average if the first available beacon is caught. Then the slave must receive one beacon packet to synchronize, send the *I'm OK* packet, wait for the next broadcast and receive confirmation in another broadcast. This should sum to 50 ms scanning, 2 ms receiving before transmitting and 69 B receiving the packets. The scanning process could theoretically have a different consumption, but the measured value was 24.4 mA which is a negligible difference considering the consumption of the full-speed running MCU. For TX time it would be again just 24 B with numbers from equations 14 and 15.

$$t_{\mathrm{RX\_19min}} = \frac{69 \times 8}{38.4\,\mathrm{kBaud}} + 50\,\mathrm{ms} + 2\,\mathrm{ms} \approx 66.38\,\mathrm{ms} \qquad \textit{(Eq. 22)}$$

$$r_{RX} = \frac{t_{\mathrm{RX\_19min}}}{19 \times 60\,\mathrm{s}} \approx 58.22\,\mathrm{ppm} \qquad \textit{(Eq. 23)}$$

$$I = r_{RX} \times 23.5\,\mathrm{mA} + r_{TX} \times 46\,\mathrm{mA} + 1.4\,\mu\mathrm{A} \approx 2.97\,\mu\mathrm{A} \qquad \textit{(Eq. 24)}$$

Calculations on equations 22, 23 and 24 give slightly increased consumption of 2.97 µA. The real consumption could be biased by a number of factors one of them being success rate of catching a beacon and delivering messages. The process of scanning and delivering a message is not faultless, as we see on a histogram on the illustration 15, and it often takes more than 100 ms.

10.5.2 Always listening slave

Always listening slave must transmit one 24 B packet every 19 minutes, and it must listen for a 30 B beacon every 100 ms. This means still the same TX time and TX ratio from equations 14 and 15 and RX time of 71 s with a ratio 6.25 %. The consumption is estimated to 1.47 mA. This should be probably used only for a device powered by a mains supply.

$$t_{\mathrm{RX\_19min}} = \frac{19 \times 60 \times 10 \times 30 \times 8}{38.4\,\mathrm{kBaud}} \approx 71.25\,\mathrm{s} \qquad \textit{(Eq. 25)}$$

$$r_{RX} = \frac{t_{\mathrm{RX\_19min}}}{19 \times 60\,\mathrm{s}} \approx 6.25\,\% \qquad \textit{(Eq. 26)}$$

$$I = r_{RX} \times 23.5\,\mathrm{mA} + r_{TX} \times 46\,\mathrm{mA} + 1.4\,\mu\mathrm{A} \approx 1.47\,\mathrm{mA} \qquad \textit{(Eq. 27)}$$

10.6 Consumption

For the measurement of the consumption of a slave of this network, I had to put few passive components to the measuring circuit to smooth the current impulses. The expected cumulative consumption is in range of units of µA, and the TX current is 46 mA. I don't have an ampere-meter of such parameters. I used large capacitor and inductance I had at my disposal. At the illustration 20, the connector JP1 was connected to the power supply set to 3.7 V, and the connector JP2 was connected to the measured board. The larger supply voltage increases the voltage reserve when the board is powered by the capacitor. The voltage reserve is proportional to the charge available at the capacitor. The board's voltage drops significantly when the RF chip transmits even with this large capacitor and higher supply voltage. The MCU froze a couple of times during the measurement. That was probably caused by an insufficient voltage when the MCU was going to sleep.

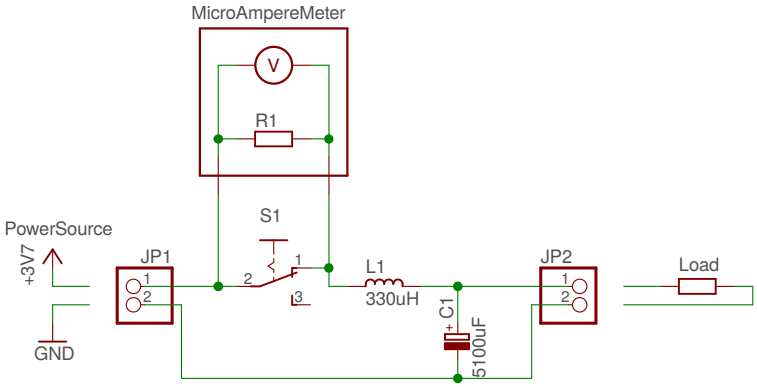

Illustration 20: Consumption, schematics

10.6.1 Slave not receiving any data

I used a multimeter set to a range of 4000 µA with resolution of 0.1 µA. The accuracy of the multimeter at this range is ±(0.1 % + 15 digits). Unfortunately, this means inaccuracy of 1.5 µA, but the smaller range is not enough to measure the current peaks. The multimeter was connected to the PC optically via USB infrared receiver which is an accessory for the multimeter. There is one of the peaks measured by the multimeter on the illustration 21.

The current peak appears each time the slave wakes up, transmits to the master and receives the response. I used a Matlab script to evaluate the measurement which found each rising edge by a simple threshold set to 10 µA. The rising edges of the peaks were

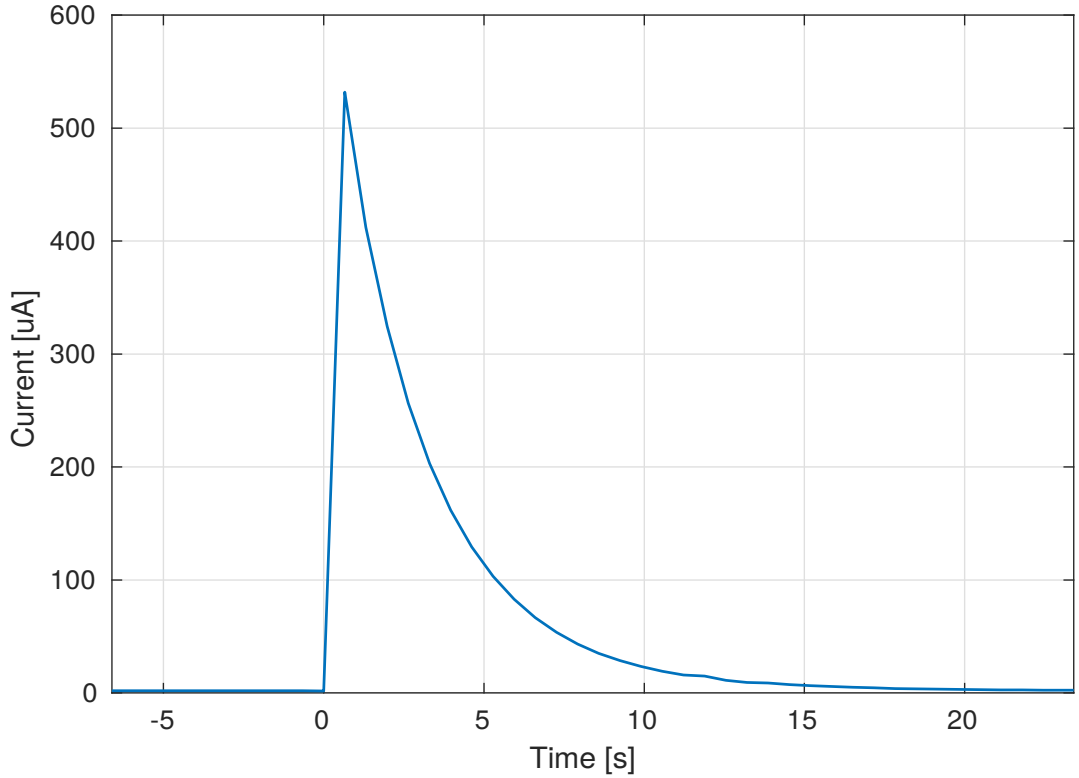

Illustration 21: Consumption, current peak

$19 \times 60 = 1140\,\mathrm{s}$ apart. Peaks that have been a different time apart were automatically eliminated from further processing. Different time between peaks was caused by pressing the button or by a frozen MCU as mentioned earlier. The peaks were used by the script, and a mean from all the current samples between one and second rising edge was calculated. The mean of exactly one period should be equivalent to the cumulative current consumption. The minimum of eleven evaluated periods was 2.35 µA, and the maximum was 6.50 µA. The mean was 4.27 µA with a standard deviation of 1.42 µA.

10.6.2 Always listening slave

Measuring the consumption of the always receiving slave was as complicated as the previous situation. That was an unexpected behaviour. The current drawn by the board at 100 ms intervals was not smoothed by the capacitor-inductance circuit. The frequency of the current at a chart at the illustration 22 was probably still larger than the sampling of the multimeter, and the results are just aliases of the real current function.

Out of 502 samples spread over 330 s, the mean was 5.78 mA with a standard deviation of 0.80 mA. The measurement was done on a 40 mA range with 0.001 mA resolution and precision of ±(0.15 % + 15 digits). Due to the aliasing, the results are quite questionable.

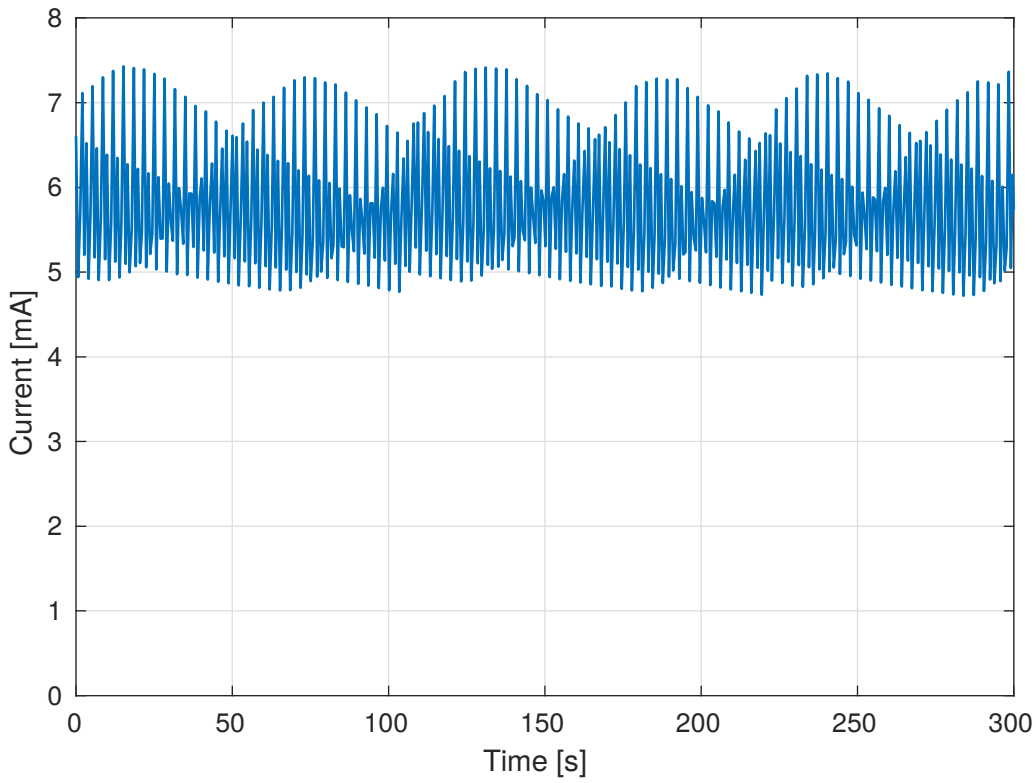

Illustration 22: Consumption, always receiving slave

10.6.3 Comparison

The results roughly fit to the estimation in chapter 10.5.1. The summary table follows:

Table 12: Consumption, conclusion

Receiving	TX only	No information	Always
Estimate, Single Frequency	2.68 µA	1.71 µA	23.5 mA
Estimate, This system		2.97 µA	1.47 mA
Measurement, This system		4.27 µA	5.78 mA

Measured consumption for the case of no information receiving slave is comparable to the estimated consumption. The standard deviation and the multimeter inaccuracy could explain the difference. The other possibility is that the larger consumption is caused by the consumption of the running MCU and receiving some time before the actual packet starts.

In the case of always receiving slave, the measurement is not too trustworthy, but the higher value could reflect that the program wasn't optimized. During the sleep the MCU has running high speed oscillator, PLL and timer to keep its synchronization. It may be possible to switch the synchronization to the low power timer and put the MCU to deeper sleep, but that would require more optimization for the particular processor.

10.7 Bitrate

To measure the bitrate, one device sent simple payloads with an id and random data. The id is 16 bit unsigned integer incremented for each payload. The other device was listening for that particular header. When the header was received, variable c was incremented by 1 to count successful transmissions. The difference between id received now and in the payload before minus 1 was added to the variable f. This was to count missing payloads that were not received successfully.

The measurement was done for a period of 100 s. As usual, no near SRD device was detected. One of the devices was running on its own, while the other device was connected to the debugger to read the resulting variables c and f.

The payloads consisted of 2 bytes of id and 6 or 30 random bytes. In total that makes length, l, to be 8 or 32 bytes of useful data. The header byte is compulsory and is not counted as a useful data.

Table 13: Bitrate, measured variables

	$l=8$		$l=32$	
	c	f	c	f
Master → Slave	14529	1478	3622	378
Slave → Master	26736	220	7937	93
Slave → Transmitting Master	9091	20	2874	6

$$bitrate = \frac{c \times l \times 8}{100\,\mathrm{s}} \qquad \textit{(Eq. 28)}$$

c is a counter of successfully received payloads, l is length of the useful data in one payload in bytes

The bitrate was calculated as a number of bits successfully transmitted on average in one second.

Table 14: Measured bitrates

	$l=8$	$l=32$
Master → Slave	9.30 kbps	9.27 kbps
Slave → Master	17.11 kbps	20.32 kbps
Slave → Transmitting Master	5.82 kbps	7.36 kbps

At the table 14 we see that the highest bitrate is for the slave transmitting 32 byte long payloads. In this case, the slave can use all slots but the beacon. The difference between 32 B and 8 B long payloads is probably given by the composition of payloads into the longer multi-slot packets, although I have expected precisely opposite results. The 8 B payloads should have better fill the longest possible packets.

Achieved speed of 20.32 kbps from slave to master or 16.63 kbps as a duplex communication is not bad. The aerial bitrate is 38.4 kbps, but we must consider that it must include preambles, syncwords, CRC and network timing. Either way, this network is not good for any kind of media, and software updates will take a long time. A firmware update 64 kiB big would take approximately a minute.

$$errorrate = \frac{f}{c+f} \qquad \textit{(Eq. 29)}$$

f is a counter of failed payloads

The errorrate was calculated as a ratio between payloads not received to number of total payloads in the transmission.

Table 15: Measured errorrates

	$l=8$	$l=32$
Master → Slave	9.23 %	9.45 %
Slave → Master	0.82 %	1.16 %
Slave → Transmitting Master	0.22 %	0.21 %

From the data in the table 15 we see that the errorrates from master to slave are quite big. This is most likely caused by the slave not receiving for a whole phase if it doesn't catch the beacon. While in the other direction, if the slave doesn't catch the beacon, then it doesn't transmit which affects bitrate but not the errorrate. Trying to resynchronize the slave by catching a packet in the next slot wouldn't make much difference. The first slot might not be used by a master, and the next slot after that might not be used at all. The addition of more and more inaccurate synchronization makes it better to wait for the next beacon and conserve power.

11 Conclusion

The created FHSS algorithm satisfies given parameters. The response time of 400 ms is satisfied in 97 % for fully sleeping slaves and even better for synchronized slaves. It was better for the STM32L0 to time the sleep by a 32.768 kHz crystal, but for other MCUs the situation might be different. The wakeup in 19 minutes doesn't need to be anyhow precise, so the input only slaves could be timed only by an internal oscillator and don't have any powered crystal oscillators at all. The consumption of 4.27 µA is a good achievement for an FHSS network considering the fast response time.

Slaves in this network can find a master, synchronize to it and are automatically put to sleep when they are not needed.

Disadvantage of this network is quite a bit of RF pollution. Four selected channels are used 1.6 % of the time. This satisfies the legislative conditions represented by the CCA rules, but it is probably larger than for most of other networks.

11.1 Quick comparison

From chapter 10.6.3 we learned that the non-hopping solution might be able to achieve consumptions of 1.71 µA, respective 2.68 µA for TX only device. This network was able to achieve a consumption of 4.27 µA. That is 250 %, respective 160 %, of the consumption of non-hopping network. The estimate for this network was a bit lower than the measured value, so the increase might be a bit lower on a real system.

The response time of non-hopping system should be negligible in the range of length of the packet which for the scenario in chapter 10.4.1 was 5 ms. Response time of this system is on average 135.2 ms which is 27 times as much.

The only important argument why anyone considers spread spectrum technologies, such as FHSS, is the interference robustness. Even when several channels become unusable, this network will have a decent chance of delivering the message. Some of the problematic channels should be eliminated completely from the FHSS selection after a longer period of time.

11.2 Tips and ideas

Chapter 10.6 is missing a measurement of a consumption of 1 s sleeping slave. The assignment specifies input only slaves with quick input response, but some applications could benefit from slaves which could be contacted in a shorter time than in 20 minutes. This kind of slave would act not only as an input peripheral but also as a slightly delayed output peripheral with perhaps increased but still very low-power consumption.

The measured value of consumption would have have any meaning at all only if the *link_slave_CC1200* layer would be rewritten to fit the LPTIM timer. Apart from that problem, the network architecture is not really optimized for such kind of slaves. The

slave in the current solution sleeps for 1 second and then receives the beacon packet. That is too much time spent in an active RX mode, and the synchronization information is not needed that often.

One possible solution might be to reserve the 9th slot for waking slaves up. The 9th slot would be almost always empty and not used by slaves. If the master needed to contact any slave, it would transmit a packet filled only with LM_HDR_WAKEUP payloads. A shorter, delayed, packet would also solve problems of inaccurately timed slaves which cannot use the last slot anyway. The receiving time of 1 s sleeping slave would be only a size of the smallest sniff of which the RF chip is capable. The imprecision of the crystal oscillator could be hidden in the length of a preamble which is sent by a master and can be as long as for the beacon packet. The slave would still have to receive the beacon packet once in range of minutes to keep the synchronization, but the consumption would be much lower than for the now used solution. Unfortunately, this idea came too late to test it.

Another disadvantage of this network is that it will probably never be able to route between different HUBs. One solution might be that sub-master would act as a slave to a different master in even phases, and it would act as a master for its own network in odd phases. This solution would create even more RF pollution, and the response times would be doubled.

Another much more complicated solution would be to use Orthogonal Frequency Division Multiplexing (OFDM) transceivers for the HUBs and simple FHSS transceivers for the slaves. With the mass production of Wi-Fi and LTE mobile chips the price of an OFDM transceiver is becoming more and more close to the price of a simple GFSK transceiver. HUBs with these transceivers wouldn’t need to transmit beacons all the time. Any slave would select any random channel and start transmitting, and the master would listen to all available channels. The other advantage would be that HUBs could be communicating with their own slaves while simultaneously routing on a number of different channels.

Used Instruments and Software

Instruments

multimeter	Tenma 72-7732A
oscilloscope	Owon MSO 5022
lab power supply	Matrix MPS-3005L-3
debugger	LPC-Link2
handheld spectrum analyser	RF Explorer ISM Combo (courtesy of Jablotron Alarms)
spectrum analyser	FSW-26 (courtesy of Jablotron Alarms)

Software

operating system	Debian / Linux	testing / 4.6.0-1-amd64
Windows-only programs	VirtualBox	5.0.24
	with Windows	7sp1
MCU development	Eclipse CDT	4.5.2
	with System Workbench for STM32	1.8, updated
	with GCC (arm-none-eabi)	5.2.1
	with GDB	7.10-1+9
	with OpenOCD (STM32F0)	0.9.0
	with OpenOCD (STM32L0)	0.10.0-dev
CC1200 configuration	SmartRF Studio	2.3.1
computational SW	Matlab Student's version	R2016a
link with oscilloscope	Owondump	0.3
link with multimeter	72-7730 Interface Program	3.00

Contents of the enclosed medium

/docs/
some of the documents listed in the bibliography

/drawings/
other illustrations used in this thesis, not created by Matlab

/eclipse/
exported MT-libnetwork and sample projects for Eclipse CDT and System Workbench for STM32

/matlab/
Matlab scripts and functions used for this thesis, with some of the measurement data

Bibliography

1: Texas Instruments Incorporated, CC1200 Low-Power, High-Performance RF Transceiver Datasheet, 2014

2: Texas Instruments Incorporated, CC120X Low-Power High Performance Sub-1 GHz RF Transceivers User's Guide, 2013

3: STMicroelectronics, STM32F072x8/STM32F072xB Datasheet, 2015

4: STMicroelectronics, STM32F0x1/STM32F0x2/STM32F0x8 Reference manual, 2015

5: ARM Limited, Cortex-M0 Devices Generic User Guide, 2009

6: STMicroelectronics, STM32L072x8/STM32L072xB/STM32L072xZ Datasheet, 2016

7: STMicroelectronics, STM32L0x2 Reference manual, 2016

8: ARM Limited, Cortex-M0+ Devices Generic User Guide, 2012

9: The European Parliament and The Council of The European Union, Radio Equipment Directive 2014/53/EU, 2014

10: Electronic Communications Committee, ERC Recommendation 70-03 Relating to the use of Short Range Devices (SRD), 2016

11: ETSI, EN 300 220-1 V3.1.0, 2016

12: ETSI, EN 300 220-2 V3.1.0, 2016

13: ETSI, EN 300 220-3-2 V1.1.0, 2016

14: ETSI, EN 300 220-1 V2.4.1, 2012

15: Bluetooth SIG, BLUETOOTH SPECIFICATION Version 4.2, 2014

16: Artem Dementyev, Steve Hodges, Stuart Taylor and Joshua Smith, Power Consumption Analysis of Bluetooth Low Energy, ZigBee and ANT Sensor Nodes in a Cyclic Sleep Scenario, 2013

17: Tyco, Technology Overview Tyco PowerG, 2016

18: William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 2007, ISBN:978-0521880688